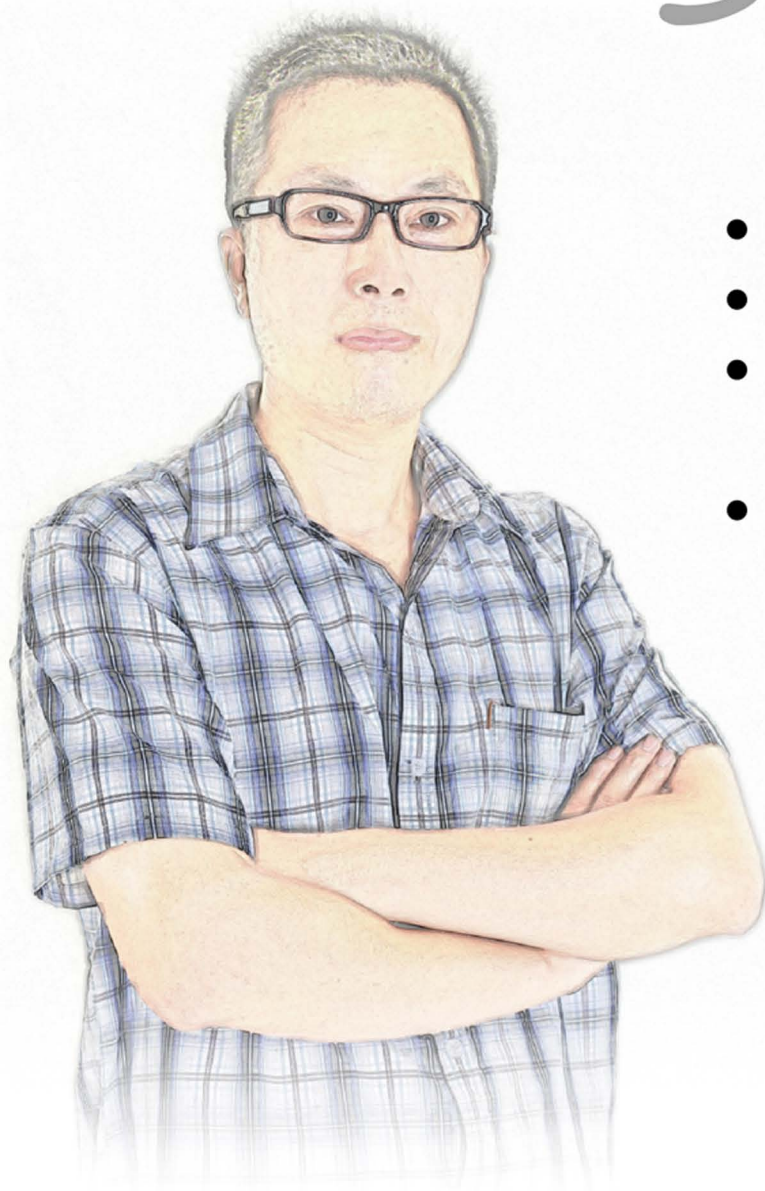


嵌入式产品分析与设计



- 注重实际能力养成
- 涵盖产品分析与设计全流程
- 全面结合了当代嵌入式系统项目开发所必需掌握的要素
- 让嵌入式产品设计变得简单

王真星 著

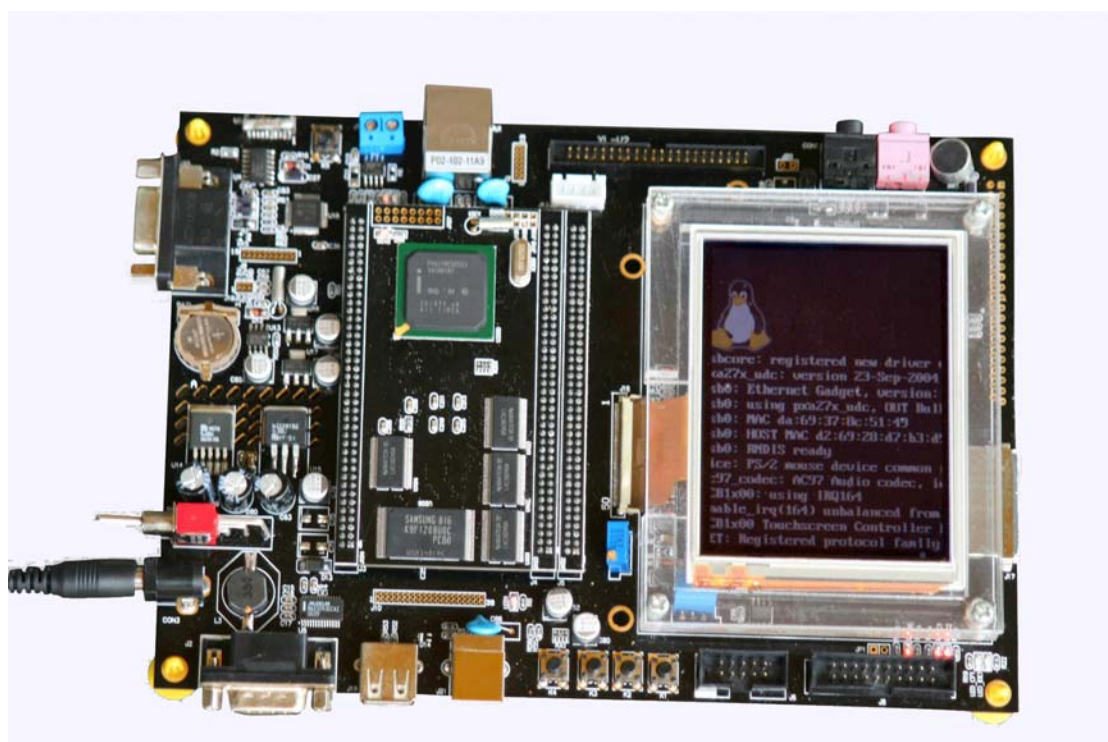


电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY

<http://www.phei.com.cn>

嵌入式产品分析与设计

王真星 著



電子工業出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书以多个嵌入式产品的开发为主线，遵循从简单到复杂的开发理念，使读者能全面了解和掌握近年来嵌入式产品开发的主要内容，具体包括软件和硬件的开发方法。本书分为两部分：第一部分包括第 1~3 章，主要介绍单片机产品开发；第二部分包括第 4~9 章，主要介绍 ARM 产品开发。

本书对工程技术人员的产品开发具有很好的参考价值。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

嵌入式产品分析与设计 / 王真星著. —北京：电子工业出版社，2013.10

（嵌入式开发直通车）

ISBN 978-7-121-20618-4

I. ①嵌… II. ①王… III. ①微型计算机—系统设计 IV. ①TP360.21

中国版本图书馆 CIP 数据核字（2013）第 120139 号

策划编辑：王敬栋（wangjd@phei.com.cn）

责任编辑：徐 萍

印 刷：三河市鑫金马印装有限公司

装 订：三河市鑫金马印装有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787×1092 1/16 印张：24 字数：614 千字

印 次：2013 年 10 月第 1 次印刷

印 数：4 000 册 定价：68.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlt@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前 言



嵌入式产品设计对实践的要求非常高，只学习理论而不实际动手，所学的知识就无法内化为自己真正的、能运用自如的技能。嵌入式产品设计要求设计人员必须熟练掌握模拟电子技术、数字电子技术、单片机技术、计算机接口技术、计算机软件技术、操作系统原理、电力电子技术、自动控制技术方面的知识，并且要有非常丰富的实践经验。要学会复杂嵌入式产品的设计，需要一步一步经历从简单到复杂，不断犯错误、不断修正错误的过程。只有走过弯路、犯过错误，才能意识到问题，才能真正掌握这方面的知识。嵌入式产品的更新非常快，技术几年一变，只有不断地研究分析最新产品，不断地学习，才能设计出符合要求的、在市场上有竞争力的产品。

本书的特点在于：特别注重读者的实际能力，通过对多个已经量产的产品的分析和设计，使工程技术人员熟悉开发稳定的、能大批量生产的产品的整个流程。为了设计先进、稳定、可靠的产品，设计产品之前就需要了解市场的同类产品是如何实现的，按照价格约束确定要设计的产品，包括如何选择元件、如何批量采购、如何布板、如何与贴片代加工厂配合，等等。元件的选择除了要了解价格，还要选择市场上的主流产品，而不要选择不容易买到的产品。如果元件太贵，则要研究能否用替代的方案降低成本。成本核算始终要贯穿于设计的全过程。除了产品稳定外，只有把成本控制在合理的范围内，所设计的产品才算是成功的。

考虑到产品的大批量生产背景，本书以消费电子产品的开发为主线，涵盖了嵌入式产品设计的主要内容，从简单到复杂，介绍嵌入式产品分析、设计和生产的全过程；从需求分析到线路图纸的设计，到 PCB 的设计，再到元件清单制作、贴片加工、软件设计、操作系统移植，直至系统在实际环境下的正确性验证，以及针对产品的知识产权保护等，对企业研发新产品具有很好的参考价值。

当前嵌入式产品开发人员的实际情况是：一部分具有硬件背景，还有一部分具有软件开发背景。对于前者，他们缺乏软件项目开发的知识，对操作系统、软件工程、面向对象、C++、网络通信、服务器、数据库等不太了解，因此在工作中不大会用到这些东西，许多还停留在 C 语言甚至汇编语言阶段；对于后者，他们不太了解计算机硬件，不会设计原理图，不会画电路板。因此，目前对软、硬件都比较熟练的开发人员较为稀缺。许多单位购买现有的评估板，然后在现成的平台上开发，最后在开发板的基础上重新布线，方案设计无法做到最优，产品自然缺乏竞争力，使嵌入式系统“量身定制”的目标大打折扣。本书的目的就在于力图改变这种局面，使开发人员同时掌握软件和硬件开发技术，开发出受市场欢迎的、高质量的电子产品。

本书的内容如下：

第 1 章 讲解与 CPU 接口密切相关的外围数字和模拟输入/输出电路的设计，是设计嵌入式产品的必备知识。

第 2 章 带领读者设计一个简单的足部按摩器产品。本章首先分析现有样品，在功能分析的基础上带领读者进行开发，包括从硬件和软件的开发到器件的选择、焊接加工等大批量生产的整个流程。

第 3 章 在掌握了简单产品开发的基础上，深入一步介绍乒乓自动发球机的开发。与第 2 章相比，这个产品用到了主机和遥控器，因此一个产品包含两部分的设计，总体难度有所提高。

第 4 章 在学习了单片机的产品开发后，要进入 ARM 的产品开发，但两者差别较大。从单片机到 ARM 并非立即就能切入，因此本章对 ARM 的相关知识进行了介绍，重点是开发环境和启动代码。

第 5 章 在具备 ARM 的基础知识以后，本章介绍了一些外设接口。这些外设是开发产品时最常用的，必须熟练掌握。

第 6 章 在对国外拖地机器人产品样品进行硬件和功能分析的基础上，讲解用 ARM Cortex 的 STM32 开发没有操作系统的拖地机器人产品。本章主要介绍拖地机器人的基础硬件和软件设计，并在最后给出了拖地机的专利撰写样例。通过对本章的学习，读者能够熟悉目前基于 ARM 控制器产品开发的大致流程。

第 7 章 讨论 Linux 下的文件系统。本章为基于 Linux 操作系统的产品开发做好准备。

第 8 章 介绍嵌入式 Web 开发的相关知识，对 Web、CGI、数据库、Socket 通信及 HTTP 编程进行介绍，并给出如何融合这些技术进行实际远程控制的方案。

第 9 章 在前两章的基础上，介绍基于嵌入式 Linux 系统的家庭智能网关的开发，同时还提供了此产品的专利申请案例。

本书的出版得到上海第二工业大学校级重点学科（培育）建设项目“计算机应用技术 XXKPY1301”的资助。感谢刘中原、石林祥、陈林为本书的出版提供的帮助，徐小方对本书写作提供的支持。这里要特别感谢我的母亲，她听到我要把这么多年来开发的产品系统性地写为书稿，认为此乃利益人民大众之事，非常支持。

本书作者的联系邮箱为 superwang2002@hotmail.com，读者也可访问 <http://www.armv.cn> 进行技术讨论。

目 录

第 1 章	把好两头是关键——输入与输出基础知识	1
1.1	输出	2
1.1.1	最简单的 I/O 输出	2
1.1.2	感性负载对 I/O 输出的影响	3
1.1.3	隔离输出	4
1.1.4	输出电压匹配	6
1.2	输入	7
1.2.1	输入上拉电阻的确定	7
1.2.2	模拟输入的问题	9
1.2.3	模拟输入的等效网络	10
1.2.4	模拟小信号处理	11
第 2 章	最简单的例子——足疗机控制器设计	14
2.1	功能需求分析	15
2.1.1	人机输入分析	15
2.1.2	运行模式分析	16
2.1.3	电路板组成部分分析	17
2.1.4	足疗控制器功能划分	20
2.2	基于成本约束的控制电路板原理图设计	20
2.2.1	成本控制相关的考虑	20
2.2.2	CPU 控制板的设计	21
2.3	CPU 控制板的 PCB 设计	22
2.3.1	PCB 及贴片的成本考虑	22
2.3.2	焊接要求	23
2.4	价格成本核算	24
2.5	控制器开关电源设计	26
2.5.1	线性电源的设计	26
2.5.2	开关电源设计基本原理	27
2.5.3	用 PI Expert 设计足疗机的开关电源变压器	30
2.5.4	开关变压器的设计	37
2.5.5	控制集成电路部分	40

2.5.6	开关电源的 PCB 设计	44
2.6	控制器其他部分设计	45
2.7	给 PCB 代工厂提交的资料	48
2.8	足疗机软件设计	51
2.8.1	红外通信设计	52
2.8.2	PWM 产生	56
2.8.3	定时扫描显示	57
2.8.4	键盘处理程序	61
2.8.5	间断运行模式程序	62
2.8.6	定位程序	64
2.8.7	主控子程序	65
2.8.8	主控程序	70
第 3 章	更进一步——乒乓发球机产品设计	74
3.1	需求分析	75
3.1.1	遥控器需求分析	76
3.1.2	主板需求分析	77
3.2	硬件功能设计和实现	79
3.2.1	落点的实现	79
3.2.2	发球个数的实现	81
3.2.3	红外接收的实现	83
3.2.4	供球电动机正反转和调速的实现	84
3.3	主控板硬件原理图设计	90
3.4	红外遥控发射硬件设计	94
3.4.1	遥控器硬件要求分析	94
3.4.2	液晶的选择	94
3.4.3	遥控器主板设计	95
3.4.4	遥控器红外发射的调制	97
3.4.5	遥控器的外观	98
3.5	软件设计规划	99
3.5.1	合理安排中断优先级	99
3.5.2	主控程序总体结构	101
第 4 章	质的飞跃——从单片机到 ARM 产品开发	110
4.1	嵌入式系统和 ARM	111
4.1.1	JTAG 接口	112
4.1.2	JTAG 标准	112
4.1.3	JTAG 硬件控制器	113
4.2	JTAG 仿真器制作	114
4.2.1	ARM 的调试结构	114

4.2.2	JTAG 仿真环境	114
4.2.3	自制简易仿真器	115
4.2.4	JTAG 仿真器硬件制作	115
4.2.5	JTAG 仿真器驱动软件	120
4.3	ADS 开发套件	124
4.3.1	在 ADS 1.2 中使用简易 JTAG 仿真头调试	124
4.3.2	ADS 中程序的调试	128
4.4	ARM 启动代码和 Bootloader	130
4.4.1	启动代码主要构成	131
4.4.2	启动代码实例分析	133
4.5	从 ADS 1.2 到 Realview MDK	140
4.5.1	工具结构的改进	140
4.5.2	分散加载文件	142
4.5.3	C 库函数的差异	143
4.5.4	开发环境迁移实例	143
第 5 章	ARM 常用外设接口	149
5.1	SPI 接口	150
5.2	模块式 LCD 的 SPI 接口设计	151
5.2.1	128×64 点阵显示器	151
5.2.2	模块引脚说明	152
5.2.3	模块式 LCD 硬件连接	154
5.2.4	模块式 LCD 内部存储器	155
5.2.5	SPI 接口 LCD 显示程序	158
5.3	SPI Flash Memory 编程	164
5.3.1	SPI Flash 硬件接口	164
5.3.2	AT45DB041 的软件接口函数	166
5.4	I ² C 接口	170
5.4.1	上拉电阻与传输速率	171
5.4.2	I ² C 总线三种信号	172
5.4.3	软件仿真 I ² C 示例	174
5.4.4	带 I ² C 硬件控制器的程序	179
第 6 章	基于 STM32 的室内导航家用拖地机	181
6.1	对 mint5200 进行拆解	183
6.2	设计方案	185
6.2.1	外观的修改	185
6.2.2	机械设计	186
6.2.3	室内导航方案的选择	187
6.2.4	导航系统方案设计	188

6.2.5	红外通信方案设计	192
6.2.6	保证直线行走的设计方案	194
6.2.7	方案的合理性分析	195
6.3	总体设计	197
6.4	硬件设计	198
6.5	软件设计	204
6.5.1	关于 STM32 固件库	204
6.5.2	异常信号的处理	204
6.5.3	电动机控制部分	206
6.5.4	红外和噪声波载波发生	208
6.5.5	PID 电动机速度控制	210
6.5.6	MPU6050 陀螺仪及姿态解算	214
6.5.7	有关清洁覆盖算法分析	216
6.5.8	规则动作库	220
6.6	拖地机产品样机	221
6.7	拖地机专利撰写举例	222
第 7 章	基于 OS 层面 ARM 必备知识——嵌入式 Linux 文件系统	227
7.1	Linux 文件系统简介	228
7.1.1	ext2 和 INODE	229
7.1.2	虚拟文件系统 (VFS)	230
7.2	注册文件系统	233
7.3	安装文件系统	233
7.4	在虚拟文件系统中搜寻文件	235
7.5	卸载文件系统	235
7.6	/proc 文件系统	235
7.7	设备特殊文件	236
7.8	常见的 Flash 文件系统	236
7.8.1	Flash 的特点	236
7.8.2	JFFS2	238
7.8.3	YAFFS 文件系统	240
7.9	根文件系统	240
第 8 章	将设备联网——嵌入式 Web Server 的实现	245
8.1	Web 基础知识	246
8.1.1	HTTP 协议	246
8.1.2	HTTP 请求	246
8.1.3	HTTP 应答	247
8.2	面向电子商务的 B/S 结构	248
8.3	Web Server 制作网页	249

8.4	CGI 工作原理	250
8.4.1	环境变量	251
8.4.2	CGI 标题和 GET/POST	252
8.4.3	CGI 程序的开发	254
8.4.4	几种常用数据库接口	255
8.4.5	几种常用 CGI 及其 Web 开发语言	255
8.5	JavaScript 脚本	257
8.5.1	JavaScript 的语句及语法	259
8.5.2	JavaScript 编程举例	262
8.6	socket 通信	264
8.6.1	TCP Socket 编程举例	266
8.6.2	UDP Socket 编程举例	271
8.6.3	HTTP 请求中 Client 与 Server 的交互过程	275
8.6.4	一个简单的 Web 服务器例子	276
8.7	嵌入式 Web 服务器 Boa 的特点	280
8.7.1	Boa 的功能实现	281
8.7.2	Boa 的移植步骤	284
8.7.3	CGIC 库的移植	286
8.7.4	HTML 模板的制作	287
8.7.5	一个综合的 Web 测试实验	288
8.8	通过网络远程控制开发板上的灯	301
第 9 章	基于 Linux 的家庭网关设计	306
9.1	产品开发背景	307
9.2	功能需求	307
9.3	家庭网关设计	309
9.3.1	网络通信设计	310
9.3.2	网关软件架构	311
9.3.3	关于视频硬件设计	312
9.3.4	系统整体的硬件设计	312
9.3.5	家庭网关系统包含的软件	313
9.4	硬件平台设计	315
9.5	嵌入式 Web 开发概述	317
9.5.1	Mizi Linux	317
9.5.2	Boa 小型 Web 服务器	317
9.5.3	SQLite 轻型数据库	318
9.5.4	网关的软件平台构造	318
9.6	软件环境搭建步骤	319
9.6.1	烧写 Mizi Linux	319

9.6.2	搭建交叉编译环境	319
9.6.3	Boa 移植	319
9.6.4	SQLite 移植	321
9.6.5	SQLite 使用方法与常用命令	322
9.6.6	建立数据库	323
9.7	CGI 程序设计与实现	326
9.7.1	CGI 与客户端的通信机制	326
9.7.2	程序中读写 SQLite 数据库	330
9.7.3	RS-485 串口读写	333
9.7.4	指令格式定义	337
9.7.5	CGI 脚本举例	340
9.8	前台网页设计与实现	342
9.8.1	HTML 简介	342
9.8.2	CSS 简介	344
9.8.3	JavaScript 简介	347
9.8.4	Ajax 技术	350
9.9	网关使用说明书	355
9.9.1	产品概述	355
9.9.2	用户登录	356
9.9.3	主界面	356
9.9.4	家电控制	357
9.9.5	系统设置	360
9.10	家庭网关产品的知识产权保护问题	365
参考文献		371

第 1 章

把好两头是关键——输入与输出基础知识

很多嵌入式学习者对 CPU 和编程很熟悉，但对输入和输出的处理并不是十分清楚，这里首先来谈谈如何正确处理输入和输出，因为输入和输出是处理器与外界交互的主要部分，处理不好会直接影响到产品性能和系统稳定性。

本章主要学习如何对输入和输出电路进行处理，输入和输出电路其实与不同的外界条件密切相关。同样的电路，在一种外界条件下是合理的，但在另外一种外界条件下却可能是不合理的。通过本章的学习，读者将掌握以下几个关键知识点：

- 三极管饱和导通式开关输出；
- 感性负载的输出；
- 高频 PWM 输出；
- 隔离式输出；
- 不同供电电压芯片之间的输出匹配；
- 普通数字输入电路；
- 模拟小信号的输入；
- 差分输入电路。

1.1 输出

1.1.1 最简单的 I/O 输出

CPU 的 I/O 输出用来给外设提供信号，或者驱动外部执行设备。因此，合理设计 I/O 输出电路是产品稳定的必要条件。输出电路的设计需要考虑驱动的功率大小、频率、噪声等因素。以下首先介绍最基本的开关量输出，条件是输出开关频率很低。图 1-1 所示为最简单的驱动电路。

图 1-1 是最简单的驱动电路，由单片机出来的 OUT 信号通过 R1 驱动 VT，R1 和 R2 的关系要按照 VT 所处的开关状态计算。也就是说，VT 只有两种状态：断开与饱和导通，不允许出现放大状态。假设电压 V_{CC} 为 10V，电阻上承受的功率超过 2W。如果三极管处于放大状态，我们假设通过调节 R1 让三极管处于 c、e 之间有 5V 的压降，这样流过 R2 的电流是 5/47 安培 (A)。三极管的功率为 0.53W，由于 8050 三极管没有散热装置，因此在短时间内将很快发热，最后发烫，通过热辐射方式散热，如果安装在封闭机箱中，很快就会烧毁。

图 1-1 中给定负载 R2，其实是要计算 R1，根据 $I_c = \beta * I_b$ ，饱和情况下，VT 的压降约为 0.2V， $I_c \approx (V_{CC} - 0.2) / R_2 \approx 0.209A$ 。假设放大倍数 β 为 60， I_b 取 3.5mA，单片机输出的 OUT 电压为 4.8V，那么计算出通过 R1 的电流是 $(4.8 - 0.7) / R_1 = 0.41mA$ 。因此图 1-1 的设计不合理，在给定负载情况下，要求通过 R1 的电流至少是 3.5mA，而实际只有 0.41mA，会导致三极管烧毁。那么如何调整呢？可以通过降低 R1 的阻值，增大 I_b 实现。现在要求至少 3.5mA，因此 $R_1 = (4.8 - 0.7) / 0.0035 = 1.171k\Omega$ ，考虑裕余量，设计 R1 的阻值为 1k Ω 。

以上的前提条件是开关动作不频繁，如果每 1s 内连续执行 10 次，则上述线路还是有问题的。问题出在什么地方呢？实际是由三极管从截止状态到饱和和状态的过程必须经过放大状态引起的。因为经过放大状态是不可避免的，所以为了降低损耗，要尽量减少在放大状态停留的时间。如果输出的频率很高，三极管同样会发热烧毁。图 1-2 所示为三极管从截止到饱和的过程，特别要注意的是其中的转换过程。

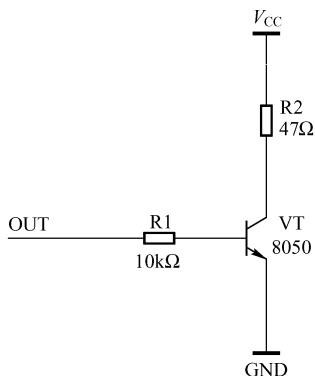


图 1-1 最简单的驱动电路

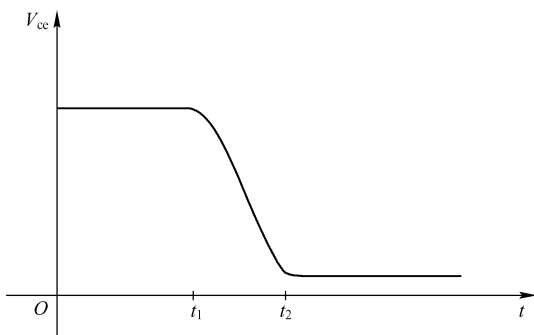


图 1-2 三极管从截止到饱和的过程

由图 1-2 可知, t_1 和 t_2 之间是三极管处于放大的状态, 其时间长短与目前三极管输出的电流大小, 输入的电流大小, b、e 之间的电容大小都有关系, 时间越长, 对输出管越不利。对于输出频率高、电流大的场合, 除了选用适合设计参数的晶体管外, 还需要在电路上进行辅助的设计, 用来尽可能减少变换的时间。也就是让信号变化期间变得陡直。

为了让信号变换显得陡直, 需要从三极管的特性进行分析。三极管的集电极载流子受基极电流的影响, 只要增大开通时的基极电流就能加快转换, 如图 1-3 所示。可以看到, 增加电容 C1 后, 当 OUT 是高电平时, 由于 C1 电压不能突变, 将迅速通过 C1 向 VT 提供基极电流, 这时的基极电流主要通过 C1 提供, 随着时间的推移, C1 充电后承受电压, 基极电流将通过 R1 提供。

以上电路虽然解决了转换时间问题, 但带来的副作用是对 CPU 输出引脚的冲击较大, 因为开通瞬间 VT 的基极回路阻抗基本是 VT 自身的 R_{be} , 一种优化的方式是给 C1 串联电阻。

图 1-4 所示是改进后的基极驱动电路, 其中, R3 降低了开通瞬间的冲击。但是 R3 的选择必须合适, 过大无法起到 C1 应有的作用, 过小会带来冲击。

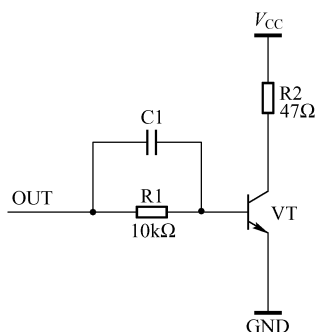


图 1-3 加快转换的方法

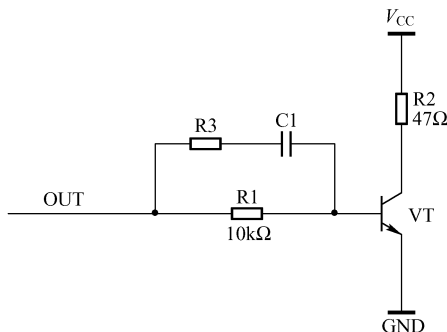


图 1-4 改进的基极驱动电路

1.1.2 感性负载对 I/O 输出的影响

前面提到的是单片机对电阻性负载的驱动问题。对于电感性负载, 情况又是怎样的呢? 图 1-5 中 VT 驱动了一个继电器, 会有什么情况发生呢?

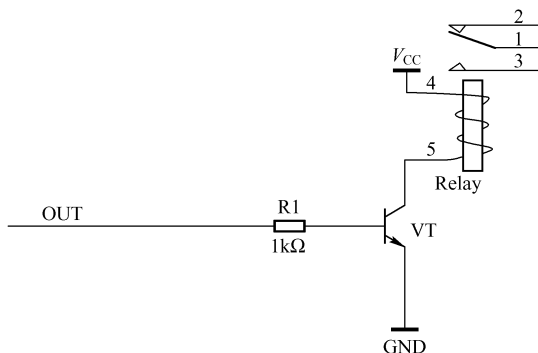


图 1-5 驱动感性负载

一开始可能很正常，但过了一段时间，不知道什么原因，VT 坏了。可能有人认为是 VT 质量不好，换一个 VT 修好了，以后也可能正常了很长一段时间，最后 VT 还是会损坏。原因就在于继电器线包其实是电感，当断开时有一个高压加在 VT 上面，非常容易击穿损坏 VT。因为电感的电流不能突变，开通的时候没有问题，电流是慢慢上升的，当 VT 关断时，电感电流不能突变，根据 $u=L \cdot di/dt$ 可知当电感量 L 一定时，VT 的关断导致电流突然消失， di/dt 很高，电压 u 也就很高，方向是上负下正，它和电源本来的电压 V_{CC} 叠加到三极管 VT 上，极有可能超过 VT 能够承受的击穿电压，导致 VT 击穿。因此，改进思路是让 VT 关闭时电感电流不会发生突变。从图 1-6 可以看到，当 VT 关闭后继电器通过 D1 续流，不会引起高压。

但是，如果 OUT 输出频率很高，图 1-6 是有问题的。

如图 1-7 所示，输出需要驱动一个直流电动机，如果直流电动机只是处于开与关两种状态，则没有问题。但如果需要对电动机进行降压调速，那么问题就来了。

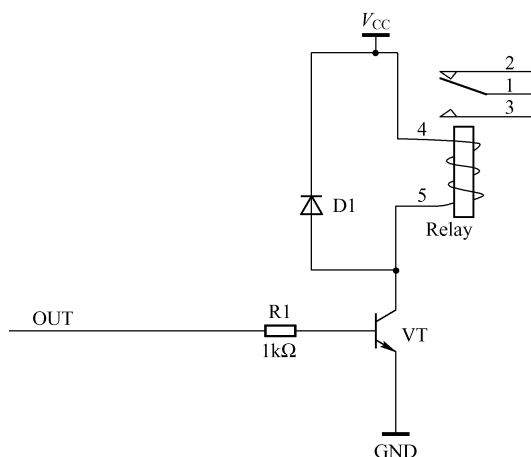


图 1-6 电感并接续流管

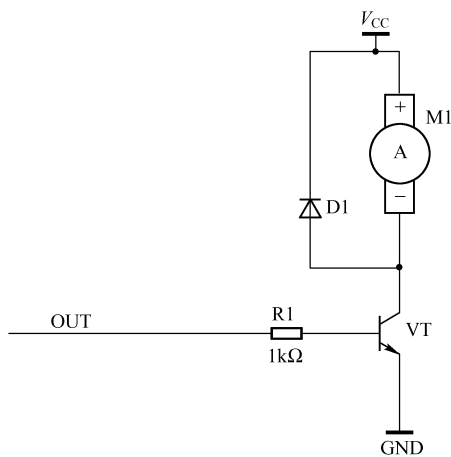


图 1-7 电动机驱动

假设电动机 M1 只是做实验用的小电动机，电流为 200mA 左右，图 1-7 是可以用的。但假设驱动的电动机电流在 1A 以上，并且 V_{CC} 是 15V，这时情况就大不一样。D1 必须仔细选择，否则会烧毁 VT。原因是 D1 若选用普通二极管，则反向恢复时间很长。假设频率是 100kHz，反向恢复时间将非常重要。当电动机在调速运行时，如果 VT 关闭，电流通过 D1 续流，然后 VT 导通，由于 D1 存在反向恢复时间，不能马上截止。这就严重影响了 VT，当 VT 导通时负载等效阻抗很小，VT 的电流很大。由于开关频率很高，这个阶段重复出现的次数很多，导致 VT 发热烧毁。为此，必须选择快恢复的二极管。在低压大电流场合，选用肖特基管，其反向恢复时间很短；在高压场合，则选择快恢复管。

1.1.3 隔离输出

当输出需要控制强电的时候，要采用隔离。根据设计要求的不同，隔离方式有光电隔

离和继电器隔离。继电器隔离主要是开关量，需要控制外设的启停场合。光电隔离主要用于频率较高且控制频繁的场所。这里主要讲解光电隔离的设计。

图 1-8 是基本的光电隔离电路。电路分为两个部分：光耦输入部分和输出部分。它们之间通过光传输信息，不存在电气上的关系。光耦输入端和普通的发光二极管驱动完全一样，一般电流为 20mA。从输出端可以看到输出用的电源的电压大小与地和输入完全不同，通过隔离方式，图 1-8 所示电路把电压从单片机的 5V 转换成 30V 的输出，也就是 Port 端的电压是 0 和 30V 两种状态。

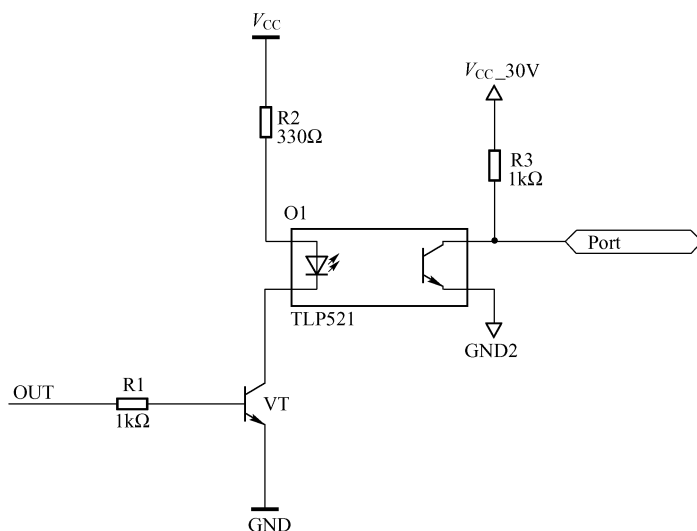


图 1-8 光电隔离基本电路

以上电路对于低频率很适合，但在频率很高的情况下就存在问题。当频率很高时，由于光耦延迟等因素，输出的波形不再完全跟随输入。

图 1-9 是光耦延迟对信号的影响。其中横坐标是时间轴，纵坐标是电压大小，下面的二维坐标表示光耦输入电压和时间的关系，上面的二维坐标表示光耦输出电压和时间的关系。当信号频率很高的时候，这个影响非常明显，会导致输出信号波形畸变。为了减小信号畸变，需要采用高速光耦，如 6N136 等。

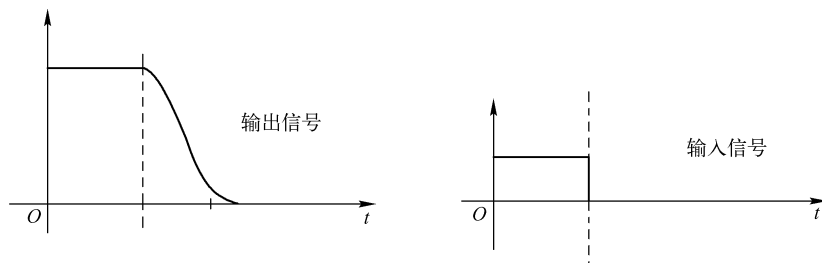


图 1-9 光耦延迟影响

在中央处理器的隔离通信应用中，处理器必须与外界电气隔离。如图 1-10 所示，通信采用 RS-485，波特率要求为 115 200bps 这里如果用 TLP521 低速光耦，肯定无法满足要求，信号会发生畸变，与远程机器无法正常收发。图中采用了 6N136 高速光耦，能顺利解决高速通信的问题。其中有一个 PC817 是低速光耦，是用来对通信方向进行切换的，也就是让通信处于发送或者接收状态，无须用高速光耦，为的是降低成本。

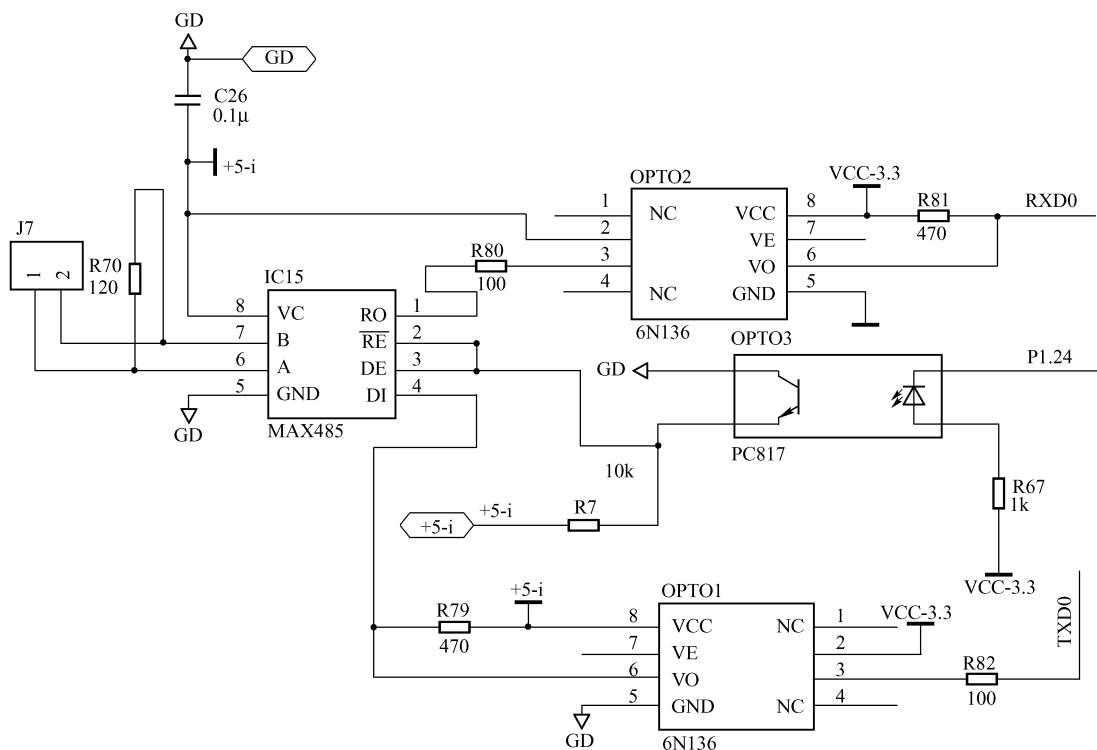


图 1-10 一个高速光耦隔离通信的例子

1.1.4 输出电压匹配

目前很多 PCB 上是 3.3V 和 5V 的器件并存，在输出时如何将两者的电压匹配呢？这里分别讲述从 3.3V 到 5V 和从 5V 到 3.3V 器件之间的连接。图 1-11 是通过三极管进行电压变换，将左边处理器的 3.3V 输出变换后，在 IC1 变为 5V 输入。如果不用三极管，也可以选用专门的电压转换芯片，原理是一样的。而 5V 到 3.3V 的转换比较容易，直接通过电阻分压即可。

输出讲完后，下面介绍输入的处理。

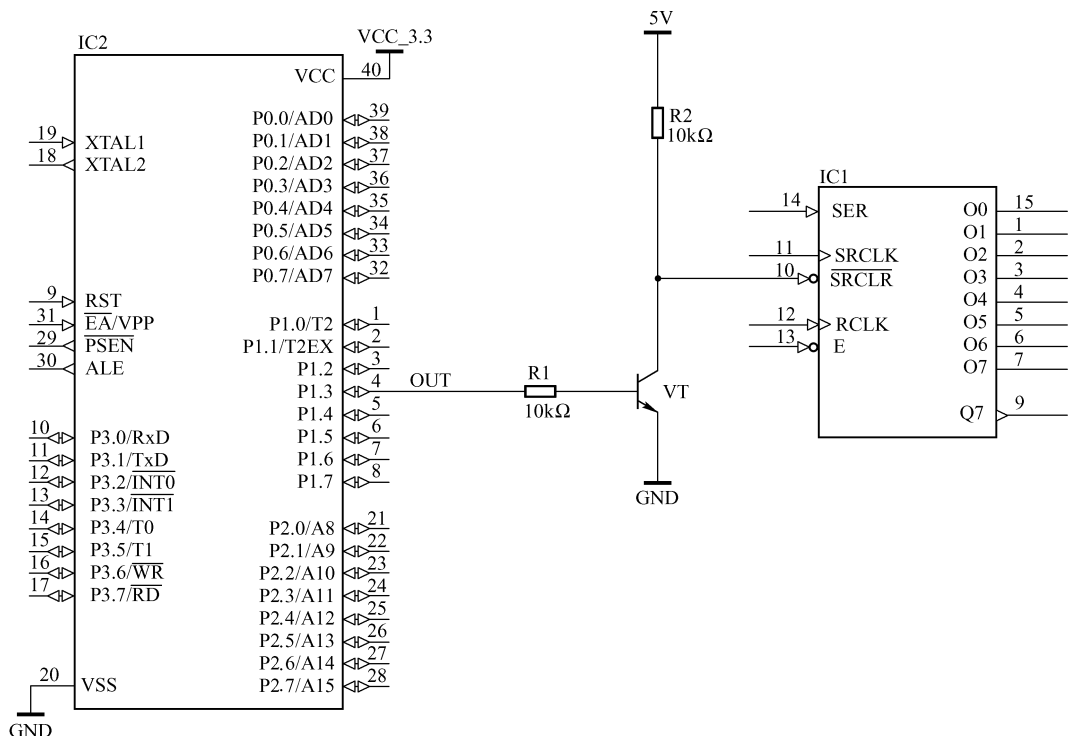


图 1-11 3.3V 到 5V 的转换

1.2 输入

1.2.1 输入上拉电阻的确定

首先介绍如何确定输入上拉电阻。假设外部中断输入，要确定输入上拉电阻。

上拉电阻和应用的环境密切相关，在干扰较小的环境下，使用 $10\text{k}\Omega$ 的上拉电阻就足够了，如图 1-12 所示。假设目前信号的流向是从外界到 CPU U4，一般外界低电平的电路驱动能力很强，而高电平的驱动能力相对比较弱。为了更加清楚地表达，我们假设极端情况即外面是 OC 门过来，完全依靠上拉电阻拉高电压。假设 p3.2 引脚由于布线原因在电路板上很长，而且绕了个半圆形，也就是说 p3.2 相当于接了一个电感。当没有干扰的时候，一切正常。低电平时， $10\text{k}\Omega$ 上拉电阻上通过的电流是 $5\text{V}/10\text{k}\Omega=0.5\text{mA}$ 。当强电磁干扰来袭时，这个干扰引起电感发电，如果发电能量很小，如 $1\mu\text{A}$ ，由于上拉电阻的吸收，不会达到输入的触发电压，当某一时刻又来了干扰，并且 OC 门恰好是浮空，发电产生的回路电流假设是 0.3mA ，则将在电阻上产生 3V 压降，导致 CPU 产生错误判断。

这时需要将 R2 变小，变为 $4.7\text{k}\Omega$ 或者 $1\text{k}\Omega$ ，减小阻值就是增加电流，让干扰信号不足以引起误动作。不过有些场合此方法还是不奏效，原因是尖峰干扰太大，在瞬间远远超过上拉电阻能吸收的电流，最终超过触发电平，导致错误动作。

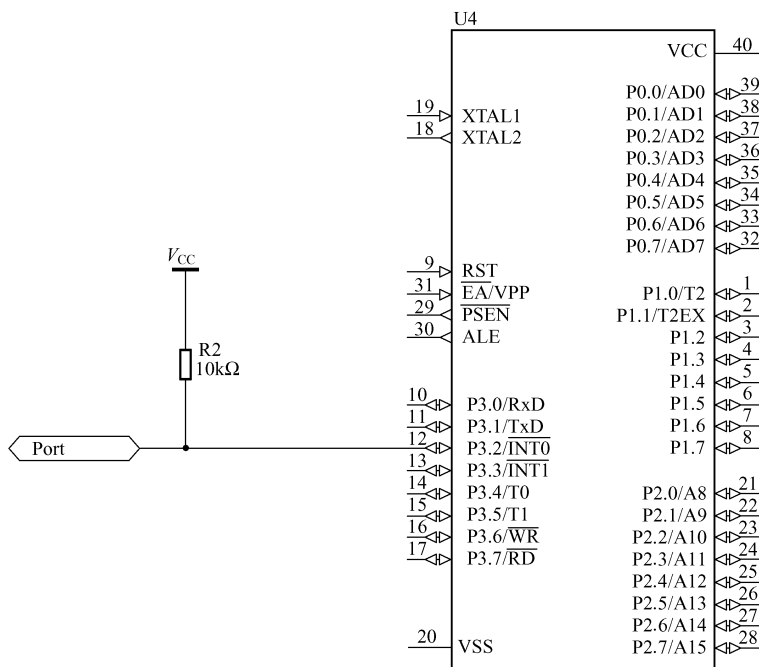


图 1-12 输入上拉电阻确定

在频率变化不大但干扰严重的场合，还可以在 $R2$ 上并接 $0.01\mu\text{F}$ 左右的电容 $C1$ ，进一步抑制干扰。如图 1-13 所示，加上电容 $C1$ 后，由于电容电压不能突变，因此能抑制强干扰。不过 $C1$ 的取值有讲究，如果信号变化频率很高， $C1$ 应取小，反之应取大。特别是在信号频率高的场合，取得太大会严重影响外界信号的波形，也会适得其反。

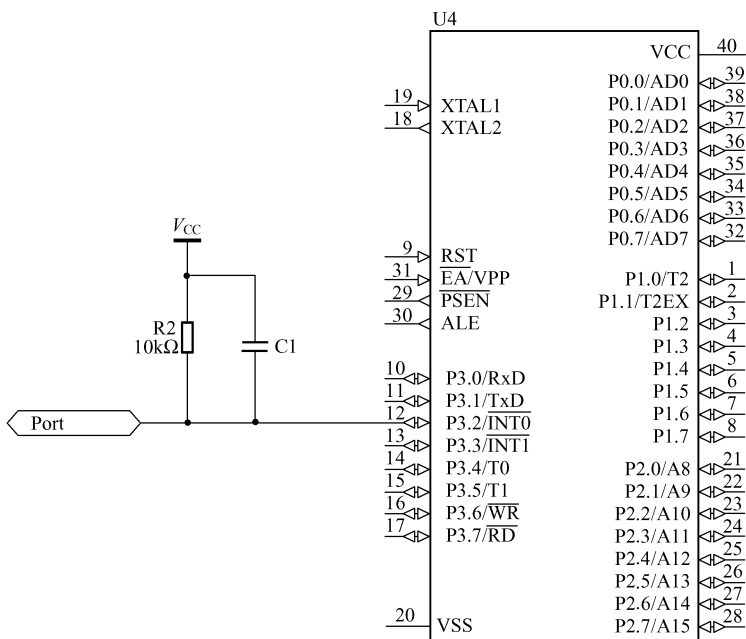


图 1-13 上拉电阻加电容

1.2.2 模拟输入的问题

CPU 需要检测外部的模拟信号，这时需要模拟输入电路。通常情况下外部模拟电压和 CPU 本身或者 A/D 转换器的电压范围并不一样，需要做模拟电压的转换，让外部电压适应 A/D 转换器的电压。例如，外部是 0~10V 的电压输入，A/D 转换器的电压是 3.3V，最简单的方法就是分压。图 1-14 中，模拟输入口是 Port，经过电阻 R1、R2 分压，在 AD4 输入端的电压范围是 0~3.3V。

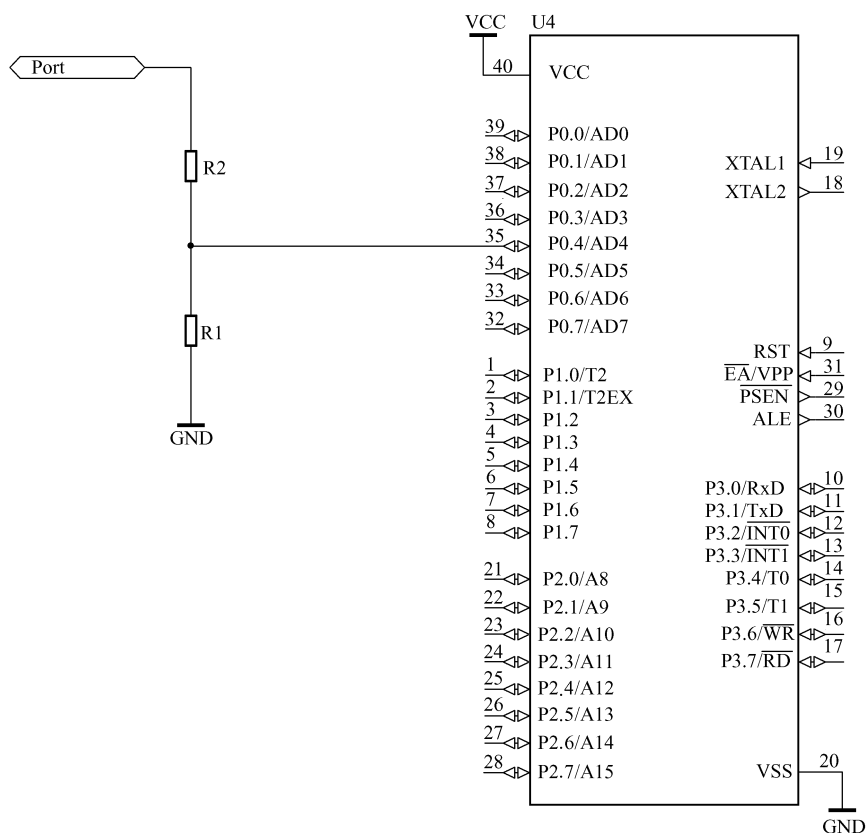


图 1-14 模拟输入分压

但是分压电阻究竟多大合适？分压电阻太小会对信号端产生影响。我们可以把 A/D 信号过来的地方 Port 看作二端口网络，如图 1-15 所示，它由电压源和内电阻构成。当 $R1+R2$ 远远大于网络内阻抗 $R0$ 时，分压的影响非常小。当 $R1+R2$ 接近内阻抗 $R0$ 时，将在 $R0$ 上产生明显的压降，影响 Port 端的输出电压。

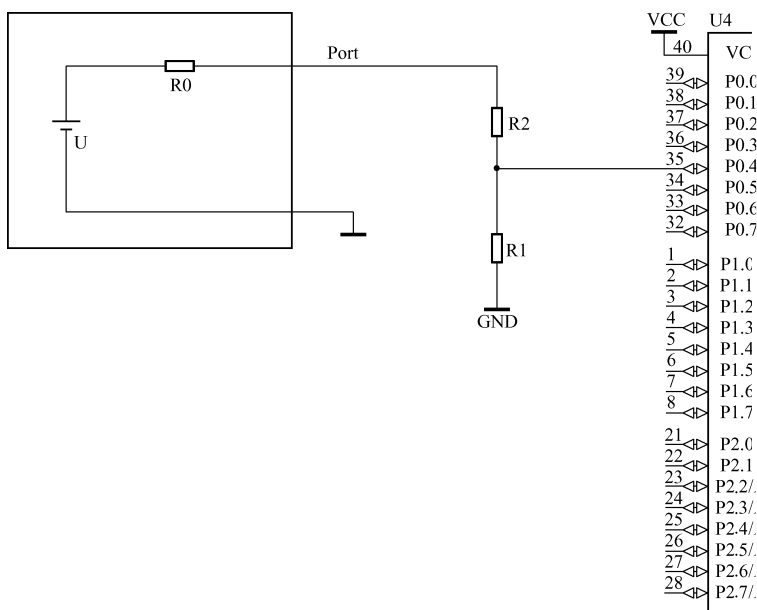


图 1-15 A/D 输入等效的二端口网络

1.2.3 模拟输入的等效网络

那么，假设 A/D 转换输入等效的阻抗非常高，怎么办？如某种气体浓度传感器前端，输入阻抗非常高，如果用一般的方法，则无法满足要求，甚至完全不能工作。这时就需要用到阻抗变换电路。图 1-16 所示基于运放的阻抗变换通过使用同相放大器实现了阻抗的变换，同相放大器的特色是能将信号由高阻抗输入变换为低阻抗输出。该电路只是一个简单的变换例子，实际中还需要对电压的范围进行限制的辅助线路。

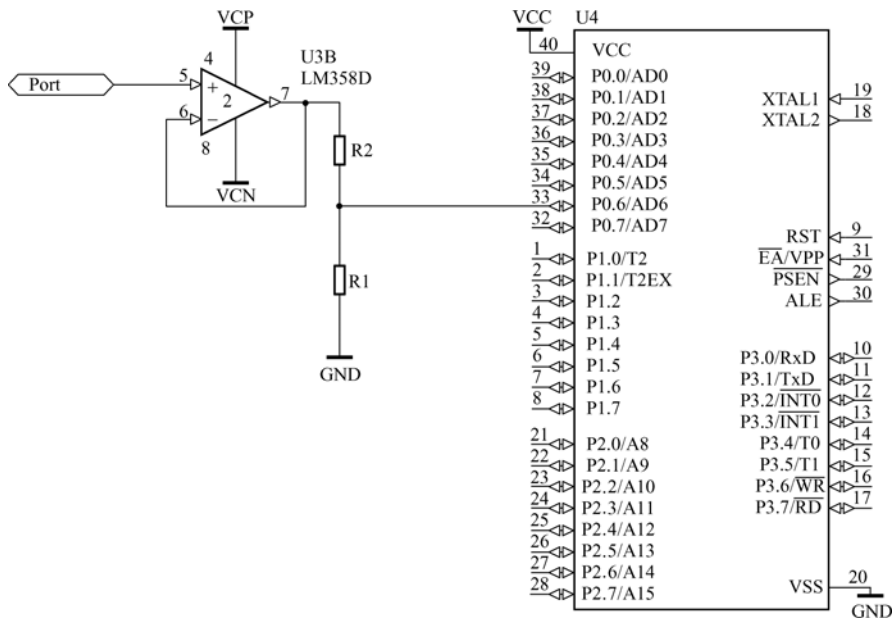


图 1-16 基于运放的阻抗变换

1.2.4 模拟小信号处理

如果输入信号很小，如最大只有 200mV，A/D 转换如何进行？此时需要先对小信号进行放大，将幅度放大到 A/D 转换输入的范围，如图 1-17 所示。

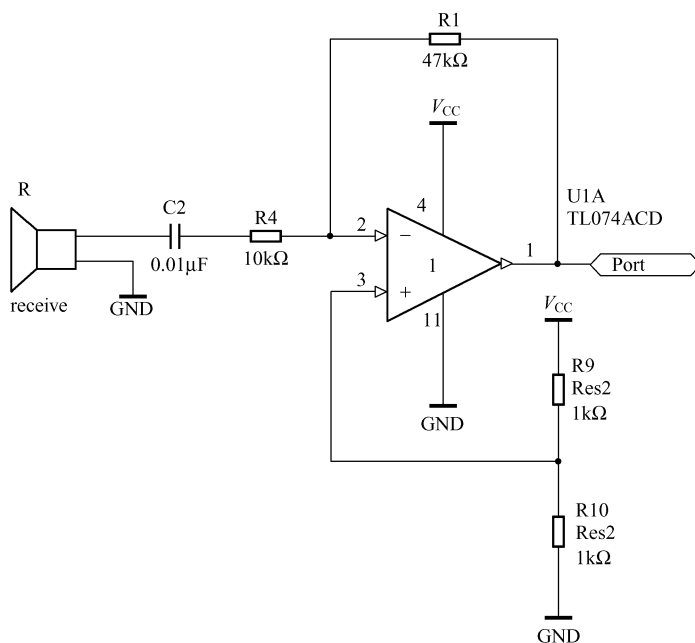


图 1-17 运放对信号放大

在图 1-17 中，假设 DSP 处理器要对外部声音信号强度进行分析，DSP 处理器通过 A/D 转换器获得模拟电压，但接收器获得的电压很小，必须通过放大器放大。图中运放的引脚 3 由于单电源供电的原因，将工作电压设置在 $V_{CC}/2$ 附近。如果引脚 3 接地，则运放应正负同时供电。一般一级放大倍数不可以做得太大，否则容易引起不稳定，应多级放大，单个放大倍数最好控制在几十倍。图 1-18 是一个三级运放构成的电路，最终输出 Port 进入处理器的 A/D 转换端口。图中运放除了放大信号外，还带有带通滤波的功能，将低于和高于一定频率的信号过滤掉。

以上谈的是对普通模拟信号的检测，如何对掩盖在噪声中的微弱信号进行处理？这就涉及差动放大器。有些场合必须用差动放大器处理模拟输入信号，最后才进入 A/D 转换器。差动放大器是一种特殊用途放大器，旨在测量差分信号，也称为减法器。差动放大器能够移除干扰共模信号，称为共模抑制 (CMR)，这是它的一项主要特性。与大多数类型的放大器不同，差动放大器通常能够测量超出供电轨的电压，适用于存在较大直流或交流共模电压的应用中。差动放大器是电流和电压监控应用的理想选择。图 1-19 是高精度的仪用放大器，差分信号从 YIN1、YIN2 输入，经过滤波，进入 AD623 的引脚 2 和 3，AD623 内部首先对输入进行阻抗变换，然后进行放大。R5 调整放大倍数，R36 调整零偏。输出 AIN2 进入 A/D 转换器。

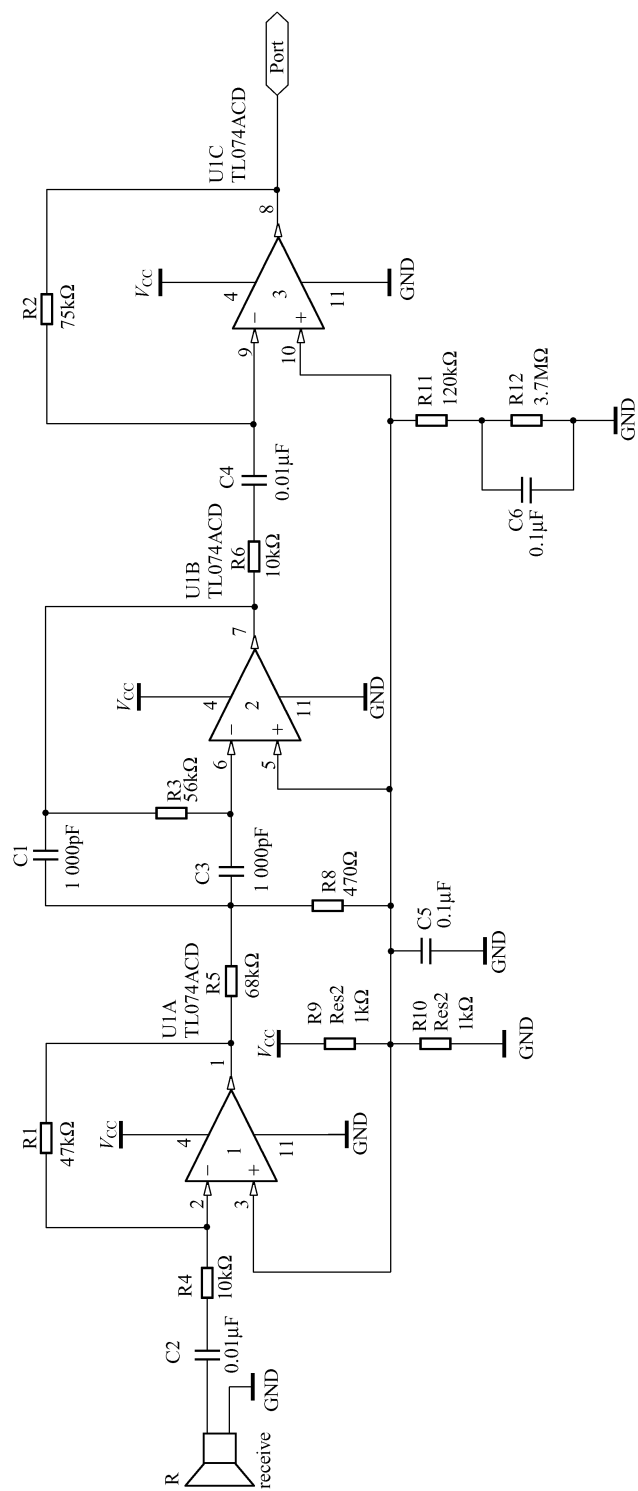


图 1-18 三级运放电路

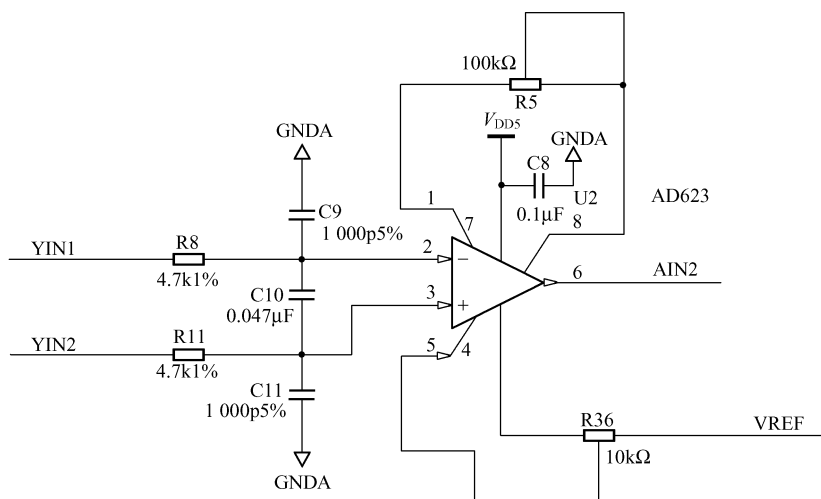


图 1-19 仪用放大器

第 2 章

最简单的例子——足疗机控制器设计

从本章开始，我们讨论最简单的产品设计——足疗机控制器的设计，如图 2-1 所示。通过这个例子，主要掌握如何分析一个已有产品的功能，利用综合的技术方法对已有产品的功能进行分析解剖，然后自己设计一套线路，能完全模仿已有的产品。



图 2-1 足疗机成品

本章主要学习的内容是：

- 线路电源部分的设计；
- 如何让硬件最少，成本最低；
- 如何让软件配合资源较少的硬件实现指定功能；
- 价格成本在设计过程中的考虑；
- 设计合理的硬件电路；
- 主要部分的程序设计；
- 代加工要准备的材料。

2.1 功能需求分析

足疗机也叫足部按摩器，是近年来市场上新出的产品，它具有加热功能和按摩功能。其中实现加热功能的是一个带电热丝脚套，方便冬天使用，按摩功能则是通过一个电动机带动按摩的轮子和塑料按摩夹板实现的。足部按摩器的控制通过按钮或红外两种方式进行。本项目通过分析已有样机的功能，重新设计一个实现方式不同、功能完全一样的控制器。

足疗机的结构非常简单，图 2-2 是其内部结构，电动机拖动齿轮后将力传递给轴承，进而带动按摩夹板运行。观察图 2-2 可以看出，夹板呈现一张一合的运动，轴承上有一个定位器，机器停止时必须让夹板张开，这样脚才能拿出来。控制器装在顶盖上。

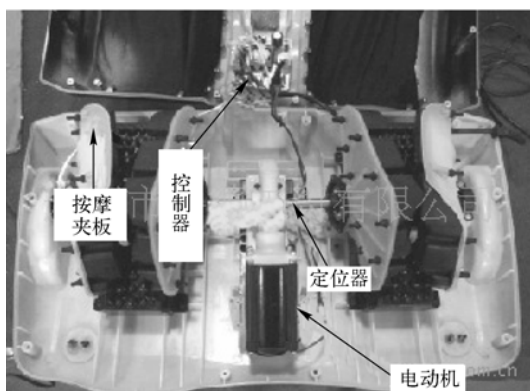


图 2-2 足疗机内部结构

2.1.1 人机输入分析

如图 2-3 所示，按摩器的控制面板共有 4 个按钮、8 个指示灯。

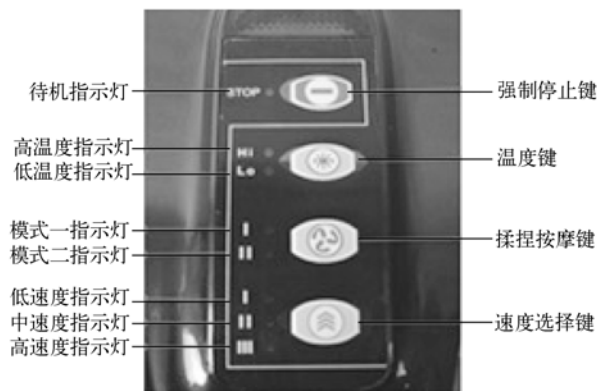


图 2-3 控制面板

指示灯从上到下分别是待机指示灯、高温指示灯、低温指示灯、模式一指示灯、模式二指示灯、低速指示灯、中速指示灯、高速指示灯。按钮从上到下分别是强制停止、温度控制、模式选择、速度选择。在控制面板内部有一个红外接收头，接收遥控器发来的命令，因此也可以通过遥控来控制按摩器的运行。

图 2-4 是红外遥控器，其按钮和控制面板上的按钮相对应，采用红外方式发射。



图 2-4 红外遥控器

2.1.2 运行模式分析

足疗机电动机的工作模式有两种，一种是持续运转模式，另一种是断续模式，也就是模仿人的按摩手法。持续模式下电动机匀速转动，带动底部的滚轮和两边的按摩夹板匀速运转。速度的高低和工作模式相互独立，互不影响。控制器主要用于工作模式的确定，具体即断续模式，用示波器观察样品的电动机驱动波形，如图 2-5 所示。

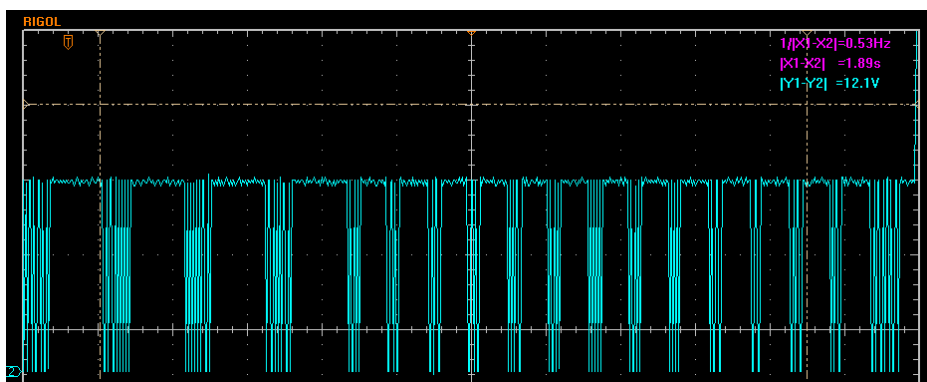


图 2-5 断续模式

图 2-5 是断续模式的波形，其实就是控制通断的时间，实现电动机的断续转动。电动机的运转速度是通过 PWM 实现的。图 2-6 是对图 2-5 的一个开通间隔放大后的波形，是一个标准的 PWM 波。断续间隔是有节拍的。

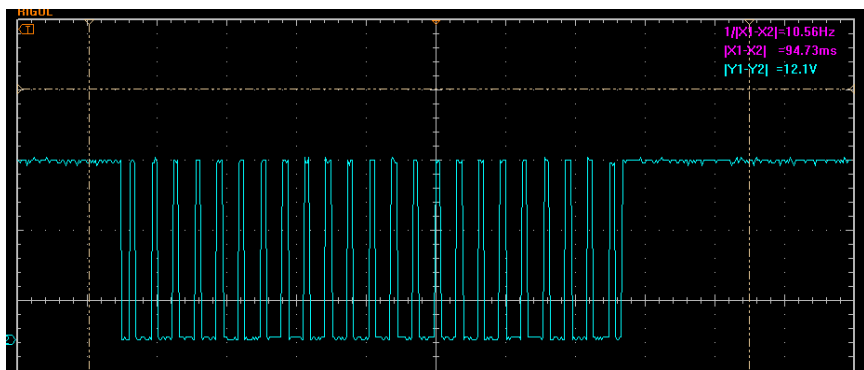


图 2-6 PWM 波

表 2-1 为断续模式节拍表。

表 2-1 断续模式节拍表

序号	右边一共循环执行 10 次	循环次数	内循环次数	开通时间（ms）	关闭时间（ms）	周期（ms）	合计（s）
1		6	7	65	135	200	8.4
2			13	30	70	100	7.8
3				800	2 400	3 200	3.2
4				800	2 400	3 200	3.2
5				800	2 400	3 200	3.2
6				800	4 800	5 600	5.6
7				800	2 600	3 400	3.4
8				330	660	990	0.99
9				330	660	990	0.99
10				800	266	1 066	1.066
11				800	1 000	1 800	1.8
总计时间 10×39.64=396.4s							39.64

以上节拍循环 10 次后自动停止，总时间大约 396s。

持续模式比较简单，只要按照速度控制 PWM 即可。

还有一个需要注意的是定位装置。电动机带动按摩夹板时，在机械机构的带动下转轴每转 360° ，夹板做一张一合的动作一次，在开机时，夹板要处于张开的状态，脚才能伸入。这就需要定位装置。定位装置是一个槽形的光电耦合器，如图 2-7 所示，转轴上有一个挡板，当挡板进入光电耦合器时，接收光电管将发生电位变化。

控制器加热输出接的是电热丝，经检测在高温和低温时输出的电压平均值分别是 12V 和 8V。系统供电分为两组，即单片机用的 5V 和电动机用的 24V。



图 2-7 槽形光电耦合器

2.1.3 电路板组成部分分析

图 2-8 是同类产品的足疗机控制器，包括电源板、控制面板和定位器，共有三块电路板。



图 2-8 足疗机控制器

图 2-9 是足疗机电源板原理图，将在本章稍后介绍。图 2-10 是 CPU 电路板的原理图。



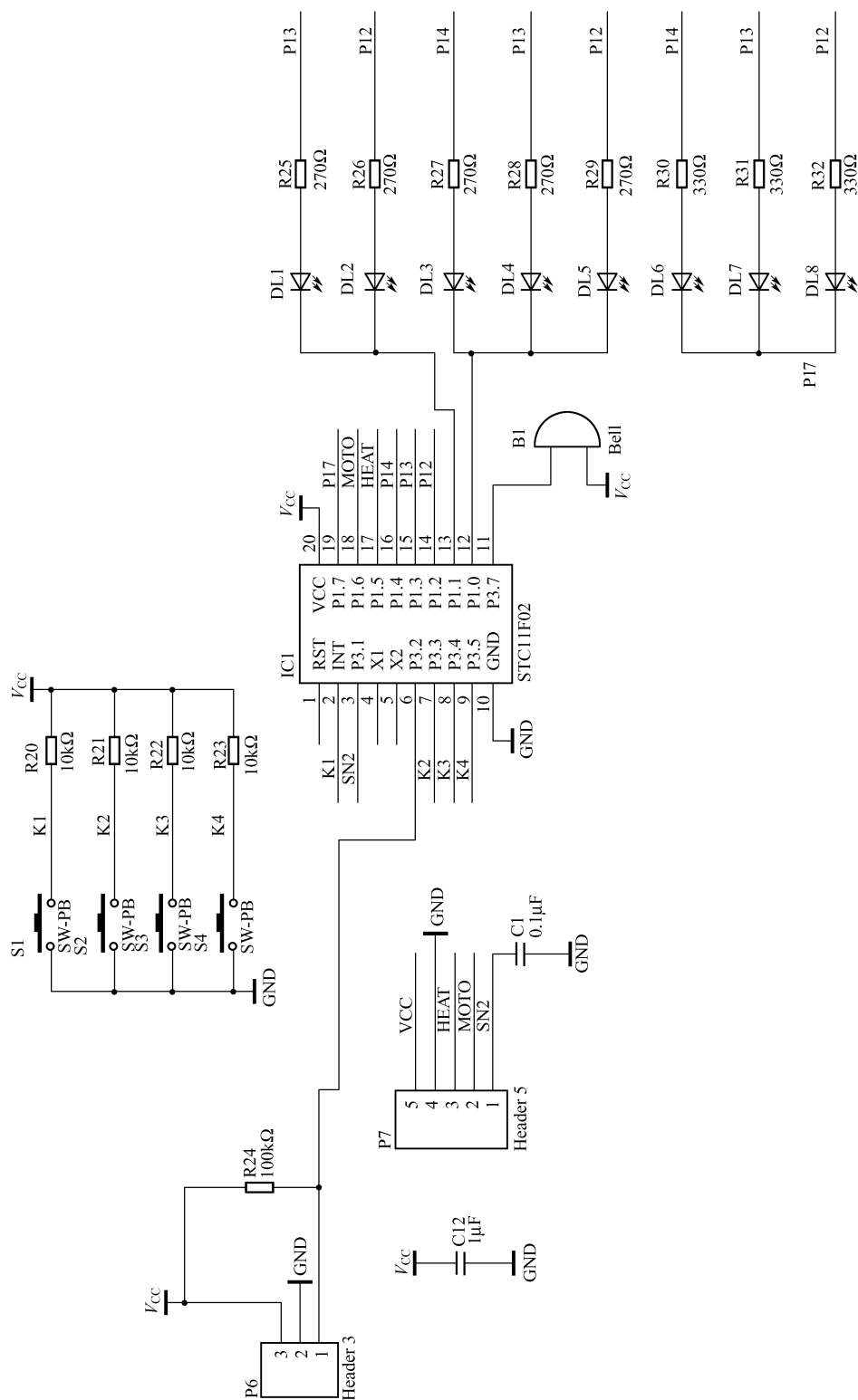


图 2-10 主控板原理图

2.1.4 足疗控制器功能划分

由样品机可知，足疗机有三个电路板，首先按照产品要求和功能为各个电路板划分功能。

产品的主要部分是驱动输出，电动机是 24V、2A 的，因此将电动机驱动和电热丝加热驱动部分放在电源板上比较合适，而控制面板上有按钮和发光二极管。CPU 既可以放到电源板也可以放到控制板上，但考虑到控制板和电源板之间连线最少的原则，建议把 CPU 放到控制板上，这样从控制板到电源板就不需要键盘和发光管相关连线，节约了成本。

2.2 基于成本约束的控制电路板原理图设计

2.2.1 成本控制相关的考虑

I/O 端口的确定：4 个按钮，8 个 LED，2 个 PWM 输出，1 个定位输入，1 个红外接收输入，1 个蜂鸣输出。按常规需要 17 个 I/O 口。按照 2011 年下半年的市场行情，此产品以每次 3 000 套批量给足疗机组装厂供货的价格是：3 套电路板+电线+遥控器+连线的总体收购价格为 32 元。注意，这 32 元还包括采购成本、贴片焊接外加工成本、检测成本、维修成本、物流费用、钢网成本、包装成本，等等。如果不控制成本，做的一定是亏本生意。另外，考虑到目前市场上抄袭严重，为了防止被其他工厂复制，要对程序进行加密，因此这里选用宏晶的 STC11F02 单片机，1 000 套的批发价格为每片 2.4 元。但此款单片机的 I/O 数量为 15 个，如果用常规的 I/O 设计，肯定不行，必须选择引脚更多的封装，当然成本也会上去。具体包括，CPU 要贵几角，电路板走线变多，带来的问题是可能存在个别 PCB 加工中的短路、开路问题，如果以后维修，同样涉及修理费。CPU 引脚每多一个贴片费用增加 2~3 分。因此，对于大批量、低利润的电子产品，必须严格控制成本，在保证质量的前提下电路板越简单越好。封装尽量采用贴片。人工焊接的问题是容易放错元件，虚焊、漏焊都可能发生。经统计，控制面板有焊盘 100 个，元件 31 个；电源板有焊盘 202 个，元件 71 个；传感器板有焊盘 12 个，元件 2 个。只要有一个虚焊，或者一个元件不稳定，整个产品就是废品。因此为确保质量，必须从以下几方面入手。

(1) 必须严格控制采购渠道，防止劣质元件进入。另外，对外加工的配件也要严格把关，本项目中一个核心元件开关变压器是外加工定做的，要保证每个变压器都能用。这就要求到对方企业考察，观察是否能保证应有的质量。

(2) 原理图设计要最优化，尽量减小不稳定性，同时要方便生产。能精简的尽量精简，要加保护的地方在成本提高不多的情况下加保护电路。这样做的目的是，虽然成本有少量的上升，但大大减少了售后服务的费用。例如，假设保护没做好，出厂的足疗机整机通过代理商进入商场，卖给了顾客，最终顾客是要返修的，这样处理的成本将大大增加，而且会给产品声誉造成不良影响。2011 年下半年的行情是，足疗机整机批发价是几百元，商场卖给顾客的价格上千元。可见，这么贵的产品，顾客对质量是非常重视的。

(3) PCB 的设计要优化，走线能粗的尽量粗，间距尽量大，过孔位置和元件焊盘之间

的距离要远,以免加工的时候短路。焊盘要尽量大,以免生产工人加工时用力过猛而脱落,也避免运输震动引起脱落。例如,功率 MOS 管个头比较大,还有一个大的散热器,共有 3 个脚固定,如果焊盘太小,运输过程中很有可能发生焊盘断裂。

另外,要严格把关贴片焊接外加工厂的质量。这其实是个两难的问题,如果要求加工质量好,故障率低,加工成本必然上去;如果压低加工成本,就一定会存在质量问题,毕竟代加工厂在控制生产工人上也要花更多成本。建议采用产品测试的方式,每个代加工的产品都进行测试,有不合格的让对方重新处理。因此要做一套测试设备。

(4) 包装处理。代加工厂可能不在本地,这样大量的货物从异地发来,中间需要 1 周的时间,经过多次物流转运,如果包装不好可能损坏很多。因此必须做防压、防震的包装。

之所以要强调质量,其实就是保证自己的利润,因为维修成本非常高。例如,一个发光二极管元件 4 分钱,人工焊接一个是 6 分,总共 1 角,如果质量不好,要维修,卸下来一个的人工费是 6 分,再拿新的焊上去的费用一共是 0.1 元,这样一来一去,原来 0.1 元的成本现在变成了 0.26 元。这只是在知道故障的前提下,查找故障呢?有可能找半天也没发现问题在哪里,这些人工成本都要考虑进去。

2.2.2 CPU 控制板的设计

下面进入设计正题,首先介绍 CPU 控制板的设计。前面已经提到,控制板要应对 4 个按钮,8 个 LED,2 个 PWM 输出,1 个定位输入,1 个红外接收输入,1 个蜂鸣输出。按常规需要 17 个 I/O 口,但目前 CPU 的 I/O 少于 17,为此采用扫描的方式来点亮 LED 发光管。如图 2-10 所示,发光管分为 3 组,每组单独把阴极接在一起,阳极分别将各组对应的脚连接起来。这样,阳极用了 3 个 I/O,阴极 3 个 I/O,一共是 6 个 I/O。比起单独的 I/O 驱动要 8 个脚,这里节约 2 个脚。LED 驱动方式为阴极分别选通,然后阳极加电压,这样 8 个 LED 轮流点亮,互不干扰。STC 单片机 I/O 内部带 MOS 驱动,因此 LED 阳极和阴极无须外加三极管,节约了成本。

STC11F02 这款不带 PWM 输出,但产品中需要用到 PWM,如果选用带 PWM 的其他型号,成本会上去,为此采用定时器的方式模拟 PWM 的发生。P1.6 和 P1.7 是模拟的 PWM 输出口,分别控制电动机和电热丝。键盘按钮一共有 4 个,分别接到 P3.3、P3.4、P3.5 和 INT。定位器的输入信号接到 P3.1,红外接收头 Header3 的输出接到 P3.2,这是外部中断 0 的输入端。程序通过外部中断来读取红外信号,进而解码出命令。按照以上思路设计的 CPU 控制板见图 2-10。

以上有些简单的地方这里还是要重复几句:

(1) 键盘按钮接了 10k Ω 的电阻到 VCC,当没有按下时,对应的 I/O 是高电平,当按下后便是低电平,程序可以知道按钮的状态。

(2) LED 必须接限流电阻,否则会烧毁。

(3) 蜂鸣器不要接限流电阻,因为其设计就可以直接接 5V 电源。有两种蜂鸣器,一种是直流、一种是交流,外观看不出来,因此采购时要注意,如果买错了是无法工作的。

(4) 红外输出引脚接了上拉电阻,因为本设计采用的是一体化的集成红外接收器,内

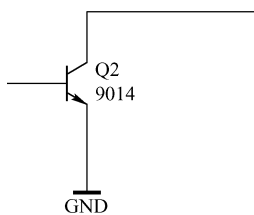


图 2-11 开漏接法

部是开漏接法（所谓开漏接法如图 2-11 所示），只有低电平，无法产生高电平，因此要外加上拉电阻，否则 CPU 的输入没有电平变化，红外也就无法工作了。

（5）P7 的第一个引脚是定位器的输入，接了一个 $0.1\mu\text{F}$ 的电容，主要用于防止开关电源和电动机的 PWM 干扰导致 CPU 误动作。大电流 PWM 的尖峰干扰加载到 25cm 长的电线上很容易产生误动作的电压，电容可吸收尖峰干扰。控制面板的 5V 电源是通过 P7 引入的，P7 连接到电源板上。

控制面板的原理图比较简单。本产品的难度主要在电源板，将在后面介绍。

2.3 CPU 控制板的 PCB 设计

2.3.1 PCB 及贴片成本考虑

目前的 PCB 一般都是双面的，本产品也可以设计为双面的。但是，单面板的成本更低，贴片加工也方便，不易出问题。单面板不存在过孔孔化不良引起的断开，不存在过孔和走线或者焊盘因为贴片工艺而短路。为此，本设计采用单面板。

以下为 PCB 代加工厂的报价，平均下来每块 PCB 的价格为每平方厘米 3 分钱，这是用 FR7 板材，其实本产品完全不必用这么好的材料，如果用其他要求低的板材，可以把每平方厘米的成本压低到 1.8 分左右。

YL-V1 中板：单片尺寸为 $10.39\text{mm} \times 3.80\text{cm}$ ，单价为 1.106 元/片， $1\,000\text{片} \times 1.106\text{元/片} + 200\text{元（工程费）} = 1\,306\text{元}$ 。

图 2-12 中的形状是根据足疗机控制板安装位置的机械尺寸决定的，从图中可以看出，电路板在底面走线，是贴片和直插元件混合的。

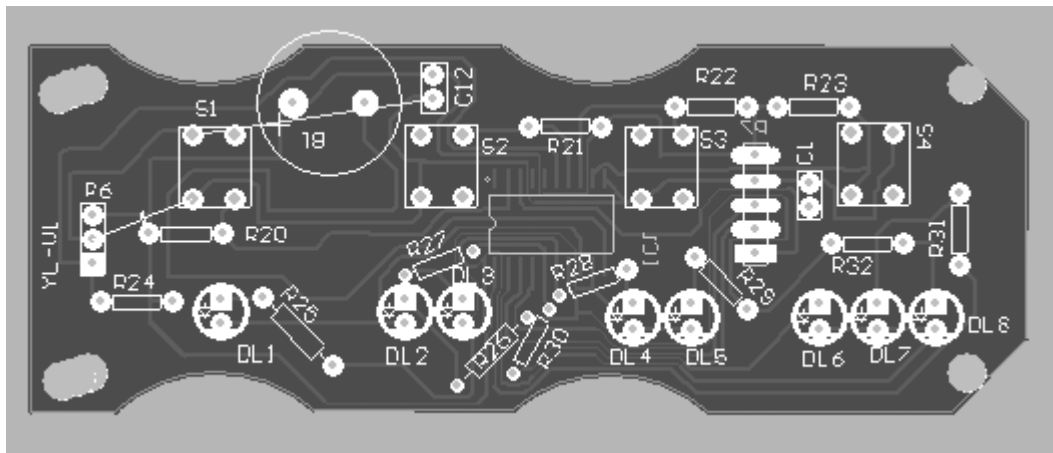


图 2-12 控制面 PCB

用直插的原因是按钮和 LED 按照产品的要求必须直插，还有红外接收头和蜂鸣器也都是直插件。此处用贴片的好处是节省焊接费，贴片封装的 CPU 价格比直插的低。按照不同代加工厂报价的不同，手工焊接基本是贴片焊接的一倍。然而，本产品在这里用贴片并不是很好，根据笔者的实地考察，很多浙江的企业不擅长做贴片，就算能贴，质量也比较粗糙，只能贴 0805 的封装，0603 的会贴不好。大部分做直插，而贴片的存在使工厂多了一道工序，不少企业表示比较难处理。浙江的代加工企业焊接报价是深圳或者上海的一半，价格上非常有优势，但客户需要自己注意质量的控制。

2.3.2 焊接要求

焊接的要求需要确定后提供给代加工厂。图 2-13 是提供给焊接代加工厂的焊接要求，前提是自己要先焊接一片样品板，并调试正常。

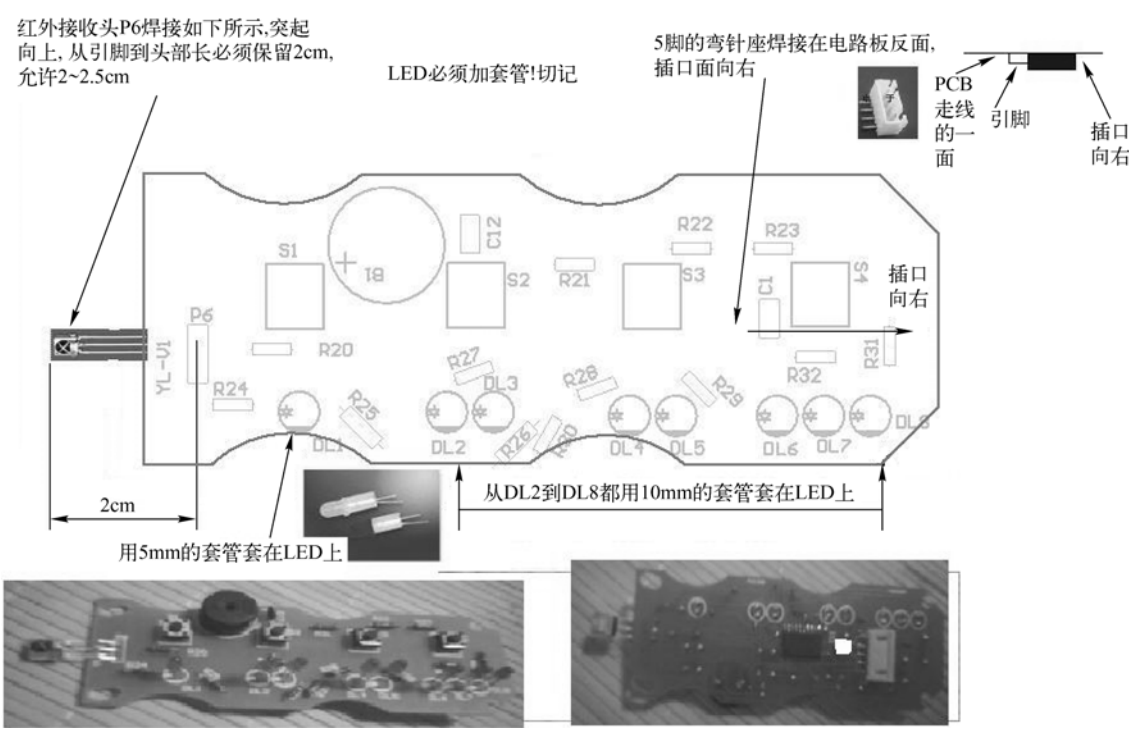


图 2-13 控制面板贴片要求

表 2-2 是和原理图对应的元件采购清单，也是焊接清单，需要提供给代加工厂。

表 2-2 显示板元件清单

名 称	标 号	备 注	数量
5V 直流型蜂鸣器	B1	12mm 直径	1
4mm 发光管套	套在 3mm 发光管上	10mm 高，套在 3mm 发光管上	7
4mm 发光管套		5mm 高，套在 3mm 发光管上	1

续表

名 称	标 号	备 注	数量
3mm 发光管	DL1, DL2, DL3, DL4, DL5, DL6, DL7, DL8	3mm, 红色, 直插 	8
STC11F01	IC1	SOP20G	1
VS1838 红外接收管	P6	直插	1
排座	P7	白色接插件 XH2.54, 间距 2.54mm, 90° 弯脚 	1
排线带头		和上面排座配套的排线, 两边带白色插头 40cm	1
10kΩ	R20, R21, R22, R23	直插	4
100kΩ	R24	直插	1
270Ω	R25, R26, R27, R28, R29, R30, R31, R32	直插	8
104F	C12	直插	2
SW-PB	S1, S2, S3, S4	6×6×6, 4 脚, 6×6 按键	4

2.4 价格成本核算

显示板贴片的价格计算如下:

贴片 20 点;

直插 4 脚的, $4 \times 4 = 16$ 点;

直插 2 脚的, $24 \times 2 = 48$ 点;

直插 3 脚的, $1 \times 3 = 3$ 点;

接插 5 脚的, $1 \times 5 = 5$ 点。

直插一共 72 点。

按照深圳普通中小型代加工厂的报价, 贴片 1.5 分一个点, 直插 2.5 分一个点, 这样控制板的代加工价格为 $0.015 \times 20 + 0.025 \times 72 = 2.1$ 元。

采购的价格如下:

贴片用的钢网 2010 年价格, 东莞为 100 元, 网络下单; 广州为 200 元, 上海是 500 元。选择东莞做钢网。

钢网是贴片加工厂用来刷锡浆的 (如图 2-14 所示), 之后贴片元件可以粘贴在 PCB 上, 放入加热炉中焊接。

XH2.54mm 线束排线, 40cm 长度, 0.14 元每个脚位, 5Pin 和 3Pin 各 1 000 根, 共 1 120 元, 此为网络下单。



图 2-14 钢网

连接线形状如图 2-15 所示。

图 2-16 是固定 LED 用的 LED 套管，一方面防止运输中 LED 弯倒，另一方面 8 个 LED 是要套入一个塑料面板中的，如果没有套管，加工后的 LED 不能保证全部为 90° 直立，工人把 8 个 LED 同时套入外壳非常困难，效率低下。

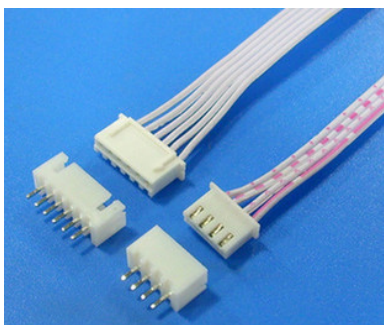


图 2-15 XH2.54mm 线束排线



图 2-16 LED 套管

套管 5×10，8 000 个，100 元；

LED 8 000 个，每个 4 分；

0603 电阻 5 000 个，15 元；

直插的 1/8W 电阻 1 000 个，15 元；

直插瓷片电容 104/50V 1 000 个，13 元；

6×6×6 直插式按键，每个 3 分，4 000 个，120 元；

蜂鸣器，0.5 元/个。

至此，显示板硬件的设计和生资料已经完成。

接下来设计电源板，电源板是本产品的核心。许多从事嵌入式的朋友只懂数字部分，对电源发生部分不懂，这样产品是不完整的。目前开关电源在嵌入式系统中的使用越来越广泛。当然，在一般情况下可以选择使用外购的电源，然而对很多产品来说必须内置电源。因此，设计开关电源也是必备的技能。

考虑到为了对开关电源有一个了解，首先介绍普通的 DC/DC 变换器，然后介绍和本设计相关的高压转低压开关电源。

2.5 控制器开关电源设计

电源设计在嵌入式硬件中占有非常重要的位置。电源设计的好坏直接影响系统的稳定性。电源设计主要从两方面来做：电源电路设计和 PCB 布线设计。

嵌入式系统中使用的电源有线性电源和开关电源两种，各有优缺点。线性电源纹波干扰小，但容易发热，效率低，体积大；开关电源的优点是效率高，体积小，发热小，其缺点是纹波大，在精密测量方面需要考虑一些特殊因素。

以下先介绍通常用的线性电源的设计，然后介绍开关电源的设计。

2.5.1 线性电源的设计

一般而言，早期的嵌入式系统主要采用 5V 供电，目前很多采用 3.3V 和 2.5V 及 1.8V 或者 1.2V 供电。例如，对于 51 单片机，一般用 5V 供电。

图 2-17 是最简单的稳压电路，输入是 12V，输出是 5V，采用的是 LM7805 稳压集成电路。

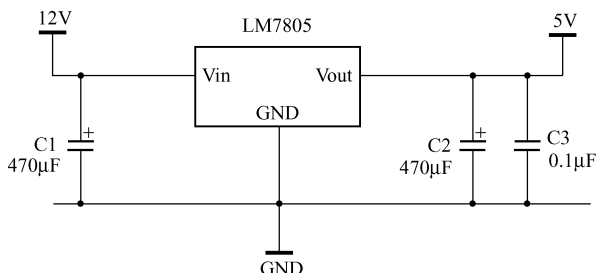


图 2-17 最简单的稳压电路

C1 是输入滤波电容，C2 是输出滤波电容，C3 用于高频成分滤波。因为 C2 是电解电容，一般的电解电容内部是一圈一圈绕制而成的，其等效电路对高频分量滤波很差，而瓷片电容 C3 内部是两个金属片直接相对安置，对高频滤波效果比较好。如果不安装滤波电容，纹波将很大。C1 可以不安装，前提是输入的 12V 直流电源内部有滤波大电容。而且外部电源到 LM7805 的引线比较短的情况，不过最好不要节省。这个电路对 9V 输入或者 7.5V 输入同样可用。其发热消耗的功率大致计算为 $P = (\text{输入电压} - \text{输出电压}) \times \text{电流}$ 。输入、输出压差越大，效率越低，发热越厉害。例如，在输出电流为 1A 时，7.5V 输入的发热约为 2.5W，12V 输入的发热是 7W。这就需要加很大的散热器，否则会烧毁 LM7805。另外，必须注意的是 C1 的耐压不能小于 12V，必须高于 12V 多一些，以避免 12V 输入中的波动及交流峰值成分。可采用 16V 耐压的电解电容。耐压再高一点也能用，但是体积和成本都会上去，没有必要。C2 则采用 6.3V 耐压的电解电容。C1 和 C2 容量的选择是，输出电流越大，容量要相对加大。如果输出在 1A 以上，一般要采用 1 000μF 或者 2 200μF 的电解电容。若要进一步降低纹波，可在输出再加电感。

图 2-18 是输出加电感的电路，这是一个 π 型滤波电路。

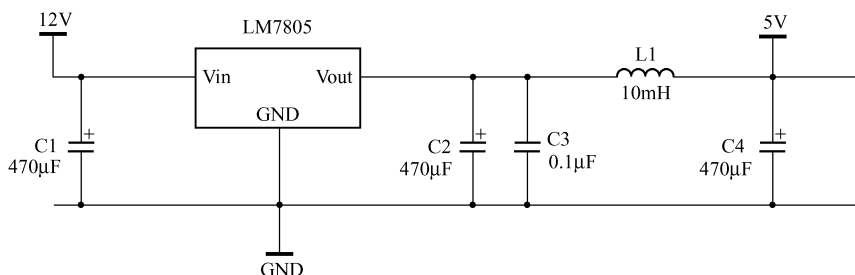


图 2-18 最简单的稳压电路+II型滤波

输出加电感能降低纹波，但是有一定的限度。如果输出电流过大，则电感的作用不但有限，而且直接影响输出电压，导致输出电压降低。因为电感内部有内电阻，因此电流在其上形成压降。

目前很多 ARM 的 CPU 都需要多组电源，如 1.8V、3.3V 和 5V 共存的系统。为此，电源的设计需要满足这些要求。图 2-19 是多组电源的电路例子，也是本书后面要介绍的 PXA270 开发板的图纸上电源部分。

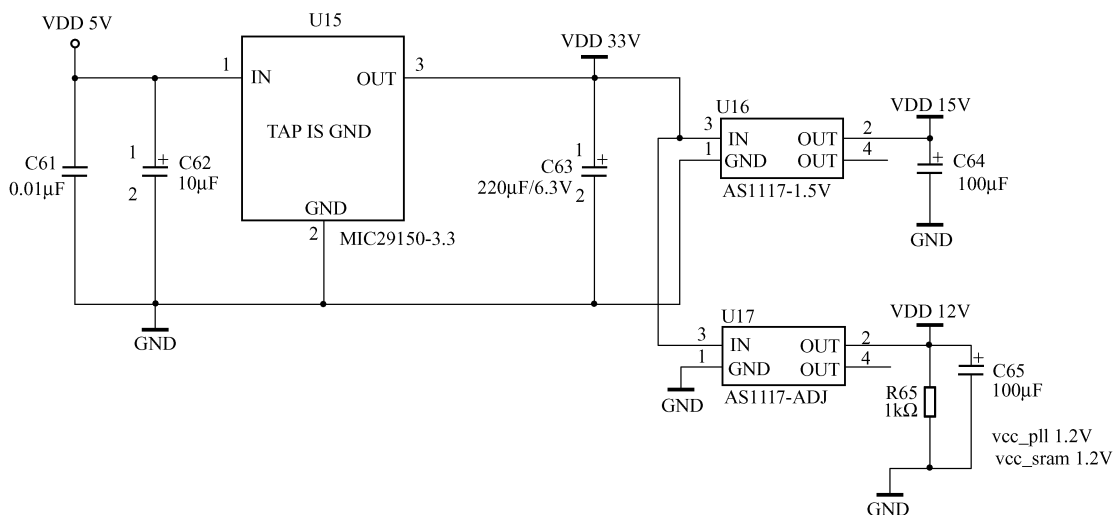


图 2-19 多组电源

图 2-19 中有 4 组电源，分别是 5V、3.3V、1.5V、1.2V（注意，由于 CAD 软件原因，3.3V 这里写成 33V，其他同理）。5V 是外部稳压输入，通过 MIC29150-3.3 生成 3.3V 电压，然后经过 U16 AS1117-1.5V 产生 1.5V 电压，经过 U17 AS1117-ADJ 产生 1.2V 电压。能否把 U16 和 U17 直接接到 5V 输入？可以，但由于压差原因，会发热。目前电路中的 U15 是 TO-263 封装，比较大，散热容易，而 U17 和 U16 采用 SOT-223 封装，散热差，但体积小。

2.5.2 开关电源设计基本原理

开关电源在嵌入式系统中的使用越来越多。一般的开关电源有两大类：升压和降压

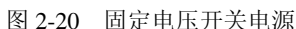
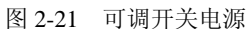


图 2-20 所示的开关电源采用 MIC4576-5.0 固定稳压开关集成电路, D13 是肖特基二极管, L3 是电感, SW 处于通断方式。当 SW 接通时, 电流由 SW 流入 L3, 电感存储能量, 同时电流也向 C60 和负载供电, 当电压上升到一定程度时, FB 反馈检测到, 断开 SW 输出。当 SW 断开时, 由于电感电流不能突变, 电流维持原来的方向, D13 导通, 电感向负载释放能量, C60 也释放能量, 当电压降低到一定数值时, U14 的 FB 检测到, 内部开始开通 SW。上面的过程重复进行。开关电源的优点是内部功率管处于开关状态, 因此效率较高。

以上介绍的是固定电压的降压电路，有时需要特定的非标准电压，这时就要用到可调开关电源。

图 2-21 是可调开关电源，其基本原理与固定电压开关电源一样，但在 FB 端加了分压电阻 R7 和 R9。分压电阻的具体大小需要按照 LM2576-ADJ 的说明书来选择，这里是 10k Ω 和 4.7k Ω ，输出为 3.8V。



升压型开关电源在嵌入式系统中的使用也比较多，特别是有关 LCD 驱动部分，经常用到升压电路。图 2-22 是升压型开关电源。

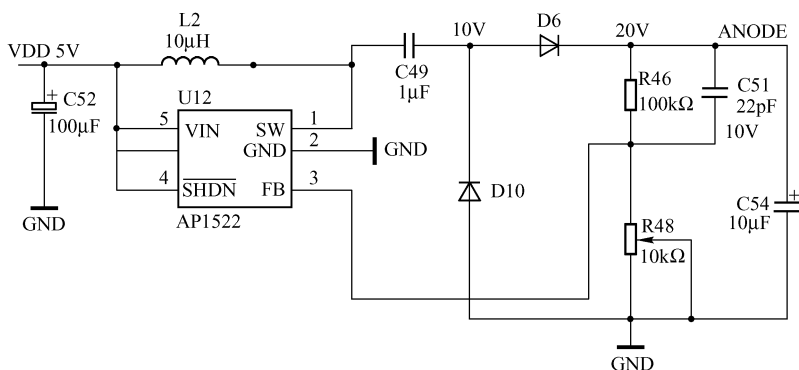


图 2-22 升压型开关电源

U12 是开关集成电路，输入为 5V 直流，输出是 20V 直流，驱动背光电路，主要用于手持设备上的 LCD 背光驱动。其工作过程如下：上电后，U12 的 1 脚 SW 和 2 脚 GND 在内部接通，电流从 VDD 5V 经过 L2 到 SW 进入地。当 C54 的电压释放到一定程度时，反馈 FB 检测到，SW 和 GND 之间的 MOS 断开，电感 L2 释放能量，电压极性是左负右正。这个电压和 VDD 5V 电压相加，出现在 SW 端，由于电容 C49 的电压不能突变，因此电压直接加到 D6 的左边，经过 D6 后出现在 D6 的右边。C54 的作用是滤波。R46 和 R48 是分压电阻，提供反馈取样。当 FB 的电压高于一定值时，SW 导通，SW 的电压接近 0V，而电容 C49 的电压不能突变，原先是左正右负，现在左边相当于接地，右边为负压，通过 D10 向地释放电流，导致 C49 的右边电压接近 0V，从而为下一次工作做准备。

图 2-23 是在图 2-22 的基础上修改形成的，可产生正负电压。

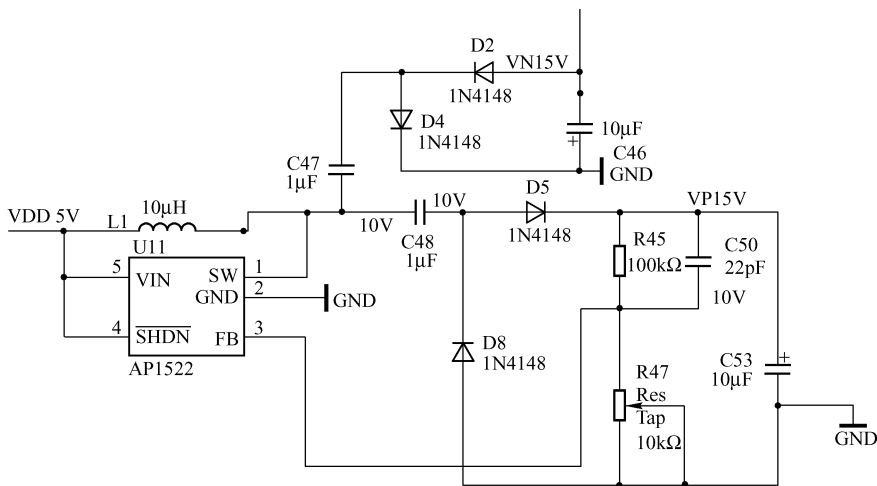


图 2-23 升压型正负电压开关电源

正电压部分的原理与图 2-22 相同，负电压的产生为：当 U11 的 1 脚 SW 和 2 脚 GND 内部导通时，C47 原来的电压是上负下正，由于 SW 接近 0V，电流从地流入 C46，经过 D2 到 C47 然后到 SW。C46 上负下正。这个过程中 C47 处于放电状态。当 SW 断开时，电流从 VDD 5V 到 L1 到 C47，到 D4 再到地，C47 充电为上负下正，为下一周期的工作做准备。

下面介绍足疗机 220V 转 24V 和 5V 的开关电源设计。

开关电源的设计核心其实就是变压器的设计，许多朋友遇到非线性变压器就头疼，确实变压器的设计非常复杂，而且一般的理论设计和实际还是有差距的，要进行大量的测试、修正才能成功。因此，要做一个稳定可靠的开关变压器，不是打样一次两次就可以完成的。开关变压器设计不好可能引起磁饱和，最终导致高压功率 MOS 管爆炸，烧毁一次回路的整流管，引起强电线路跳闸！所以试验开关电源有一定的风险，要注意安全。在初次试验时，人员要远离开关电源，在远处上电，防止炸机。

开关电源的变压器计算在电力电子技术中有详细的讨论，不熟悉也不要紧，目前有软件可以协助设计，因此没有经验的朋友也能设计出符合要求的开关电源。美国 PI 公司的软件 PI Expert 就是一个很好用的工具。

2.5.3 用 PI Expert 设计足疗机的开关电源变压器

首先在 Power Integrations 公司的网站 <http://www.powerint.com/> 下载 PI 软件，安装完成后启动软件，如图 2-24 所示。

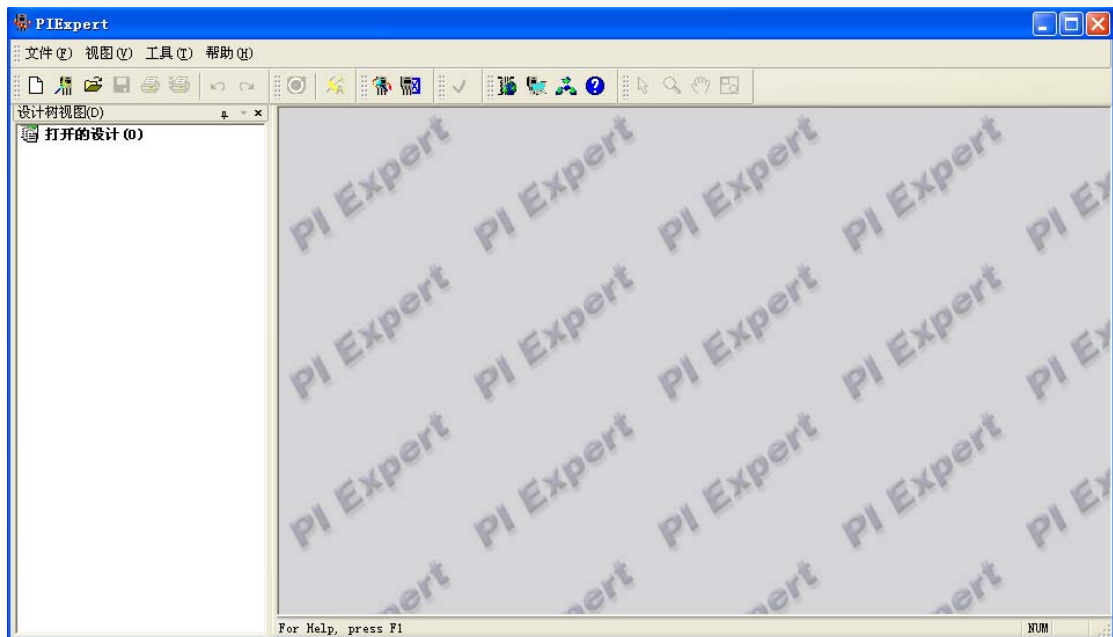


图 2-24 PI Expert 启动界面

然后新建文件，在图 2-25 所示窗口中出现几个开关电源的关键选项。

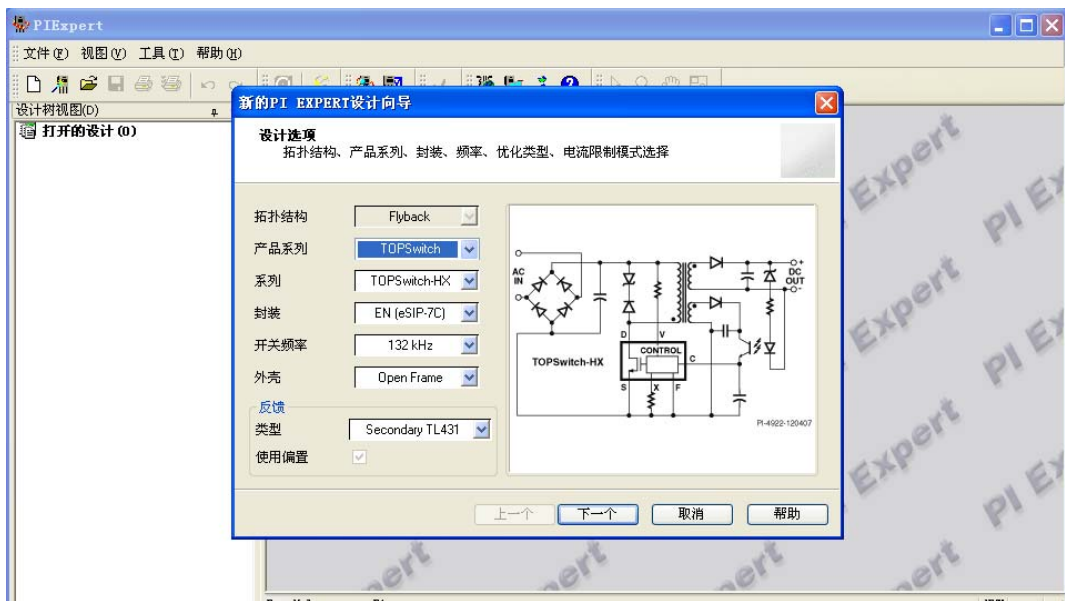


图 2-25 开关电源的选项

此处“拓扑结构”是反激式，为固定选项。反激式在中小功率的开关电源上运用较普遍，其特点是结构相对简单，容易调试。“产品系列”，在此选择 TOPSwitch。由于本产品的设计要求是电动机的额定输出电压为 24V，电流为 2A，电热丝的最大功率是 12W，单片机的功率为毫瓦级，可以忽略，因此开关电源的总功率要大于 60W，必须选 TOPSwitch。当然，设计的最后我们只用到了变压器本身，不会用 TOPSwitch 元件，原因很简单，用了 TOPSwitch 肯定亏本。TOPSwitch 对应的集成功率 MOS 的价格在 10 元左右，而用 uc3842+4N60 的价格是 2 元以内。根据输出功率的不同，TOPSwitch 元件系列分为贴片和直插，输出功率大的采用直插封装，输出功率小的采用贴片封装。它是一体化的功率集成电路，不需要外加波形的激励。资料中介绍其具备完整的保护功能，可有效防止内部的功率 MOS 在异常情况下烧毁。但是，笔者在进行实验时依然有炸开的情况。“封装”，这里选择 TO-220 封装。此处选择和电源输出参数最接近的 TOPSwitch 开关元件的目的就是让变压器尽可能接近设计要求。

在图 2-26 中可以看到，“开关频率”选择了 66kHz。还有一个选项是 132kHz，区别是在相同输出功率下 132kHz 的变压器可以选择尺寸比较小的材料，材料成本便宜，还有就是绕组也可以减少差不多一半，因此变压器的价格比 66kHz 的低不少。如果和 66kHz 的有相同尺寸，132kHz 的输出过载能力比 66kHz 的要强很多。但是，132kHz 存在的问题是发热，在相同的输出功率下，132kHz 的频率比 66kHz 的高一倍，其引起的功率 MOS 管发热也差不多高一倍。另外，由于频率的升高，功率 MOS 的吸收回路也发热严重，因此要按照输出功率、产品价格约束来综合考虑选用哪种方案。本产品的设计在样品阶段同时采用了两种方式做试验，最后发现发热的确是一个严重的问题，因此最终决定选择 66kHz 的方案。

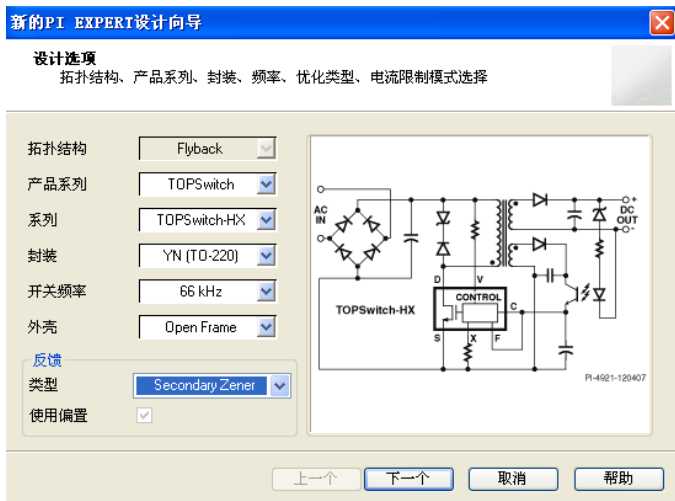


图 2-26 反馈选项

“外壳”选项中有 Open Frame 和 Adapter 两种，前一种是指开放空间，后一种是指密封盒子，其实考虑的就是散热问题，散热和变压器参数有密切的关系。在此选择 Open Frame 方式。

反馈类型有 4 个选项：初级电阻、初级稳压管、次级 TL431、次级稳压管。前两种方式稳压效果不好，存在比较大的误差；后两种方式中 TL431 稳压精度高，一半都选择这种。不过稳压管方式也未尝不可，经过测试，稳压管方式可以在本产品上稳定工作。

接着进入下一步，如图 2-27 所示，选择输入方式。由于本产品是直接接在 220V 交流上的，因此选择交流默认 85-265V 选项，这样产品在 110V 和 220V 的国家都可以使用，有利于出口。



图 2-27 输入电压选择

然后进入下一步次级输出参数设置界面，如图 2-28 所示。



图 2-28 次级输出参数设置界面

单击“添加”按钮，在图 2-29 中可以设置电压、电流。一般而言，开关电源可以输出多个电压，如 24V、5V、12V 等。在本产品中，单片机用了 5V 电压，因此需要 24V 和 5V 两个电压，这样变压器就存在两个次级绕组。次级绕组增加后成本也增加了。因此只设计一个 24V 输出，5V 的输出可通过 78L05 由 24V 降压获得。78L05 的价格为 0.2 元，相比变压器成本的上升，节约了不少。

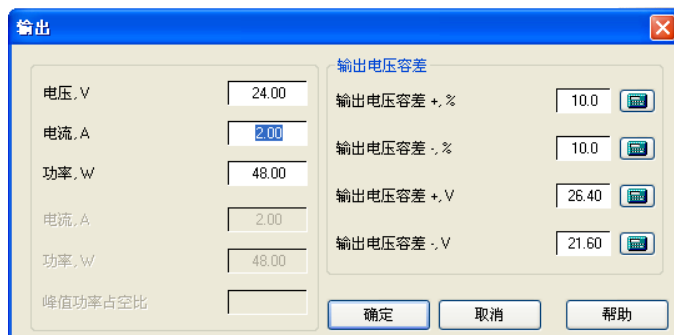


图 2-29 电压、电流设置

从图 2-29 中可以看到，我们设置的输出电压为 24V、电流为 2A，输出电压容差用默认的 10%，也就是输出电压的动态范围为 26.4~21.6V，如此大的电压降足以使电动机速度明显下降。如果减小电压容差，电动机速度能维持比较满意的水平，但变压器体积将变大，输入/输出整流的电容也变大。为满足成本约束，这里选用默认的参数。还有一个需要考虑的问题是足疗机的断续工作模式。从断续模式的波形图（见图 2-5）和断续节拍表（见表 2-1）

可了解到, 最短 335ms 电动机就要通断一次, 这样冲击电流非常大, 就不是普通的 2A 可以支撑的。因此, 开关电源设计的最大难处就在于此。经过测试, 冲击电流大约为 6A, 也就是说开关电源必须在短时间提供 6A 的冲击电流。要解决此问题, 有以下方法可选择:

(1) 开关电源的输出电流设置为 6A, 这样开关变压器就很大。

(2) 适当增加开关电源的输出电流, 然后加大输出电容容量, 让其在短时间内提供冲击电流。

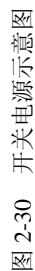
(3) 适当增大开关电源的输出电流, 让其在短时间内提供冲击电流, 开关电源具有磁饱和的风险。

方法(1)导致成本增加, 体积变大; 方法(2)对短时间的冲击能够进行处理, 针对电动机这种大功率器件, 持续冲击工作模式效果不大; 剩下的便是方式(3), 让变压器承担风险。变压器在生产的时候, 适当加大气隙, 可以防止磁饱和。经实际测试, 方式(3)是比较满意的方案, 实际中并未出现磁饱和情况。最终的输出图见图 2-30 开关电源示意图。

由于最终在产品上不会用到 TOPSwitch 器件, 因此有必要对图 2-30 进行修改。图纸左边是交流输入, 首先经过熔丝 F1 和负温度系数的电阻 RT1。在启动瞬间, 由于电解电容容量大, 会有比较大的冲击电流, RT1 可以有效阻止冲击电流。正常工作时, RT1 的电阻引起温度上升, RT1 的电阻值开始变小, 因此消耗的功率很小, 不会发热。然后是 C1 进行 EMI 滤波, C1 要选取耐压 275V 的。R1、R2 的作用是放电, 防止电源拔掉后 C1 的电无法及时放掉, 人体触碰插头会被电击。放电时间的计算公式为 $0.707RC$, 将 $R=2.2M\Omega$ 、 $C=330nF$ 代入, 计算出 $t=0.5s$ 。L1 主要用来过滤共模干扰, 本产品要求不高, 可以去掉。桥堆 BR1 整流后由 C2 滤波, 整流管选用 1A 的就可以了, 1N4007 足够。电容在这里不需要用 150 μ F、400V 的, 这个价格非常贵, 不适合用在本产品中, 选用 33 μ F、400V 的电容就可以了。功率 MOS 的吸收回路用了 R3、R4、C3、VR1。

VR1 对瞬间尖峰吸收效果明显, 而 R3、R4 和 C3 用来吸收变压器初级开关管断开时的绕组感生电压。如果不吸收, 电压会超过功率 MOS 的最大承受电压, 击穿功率 MOS 管。经实际测试, VR1 去除不会有太大的问题。C6 也是用于 EMI 滤波。如果 C6 不接, 可以正常工作, 但用示波器观察, 可发现尖峰干扰比较严重, 经常出现单片机输入端口发生干扰、导致误动作的现象, 因此必须加此电容。此电容是安规电容, 与普通电容不同, 它直接涉及人的安全。试想, 如果电源输出是 5V, 通常可以直接用手去触摸, 但如果电容 C6 发生故障, 如短路会怎样? 高压电就直接引到了低压端, 非常危险。因此需要用安规电容。

D2 是为偏置用的超快速整流管, D3 为输出用的超快恢复整流管。这里必须用超快的, 因为工作频率非常高, 若用普通的快恢复管会存在较长的反响恢复时间, 引起发热。C7 和 R8 是吸收尖峰脉冲的。C8、C9、C10 是滤波电容, C9、C10 要选低 ESR 的电解电容, 否则滤波的效果不好。它们和 L2 及 C11 构成了 II 型滤波电路, 在本产品中无须使用如此精细的滤波, 因此可去掉 L2 和 C11。R12 和 R13 是反馈回路, 分压计算为 $24 \times R_{13} \times 1 / (R_{12} + R_{13}) = 2.49V$, 分压电压输出到 TL431 的控制端。TL431 的内部原理可以用图 2-31 的功能模块示意。



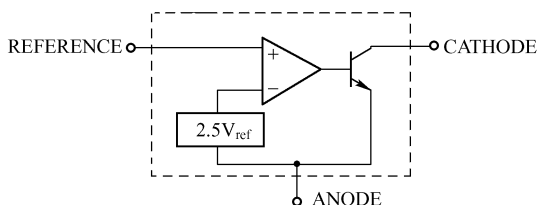


图 2-31 TL431 内部原理

由图 2-31 可以看到，VI 是一个内部的 2.5V 基准源，接在运放的反向输入端。由运放的特性可知，只有当 REF 端（同向端）的电压非常接近 VI（2.5V）时，三极管中才会有一个稳定的非饱和电流通过，而且随着 REF 端电压的微小变化，通过三极管的电流将从 1~100mA 变化。当然，该图并不是 TL431 的实际内部结构，但可用于分析理解电路。

图 2-32 中 TL431 的阴极连接到光电耦合器，R9 为限流电阻，R10 和 C12 用于提高反馈电路的动态响应。

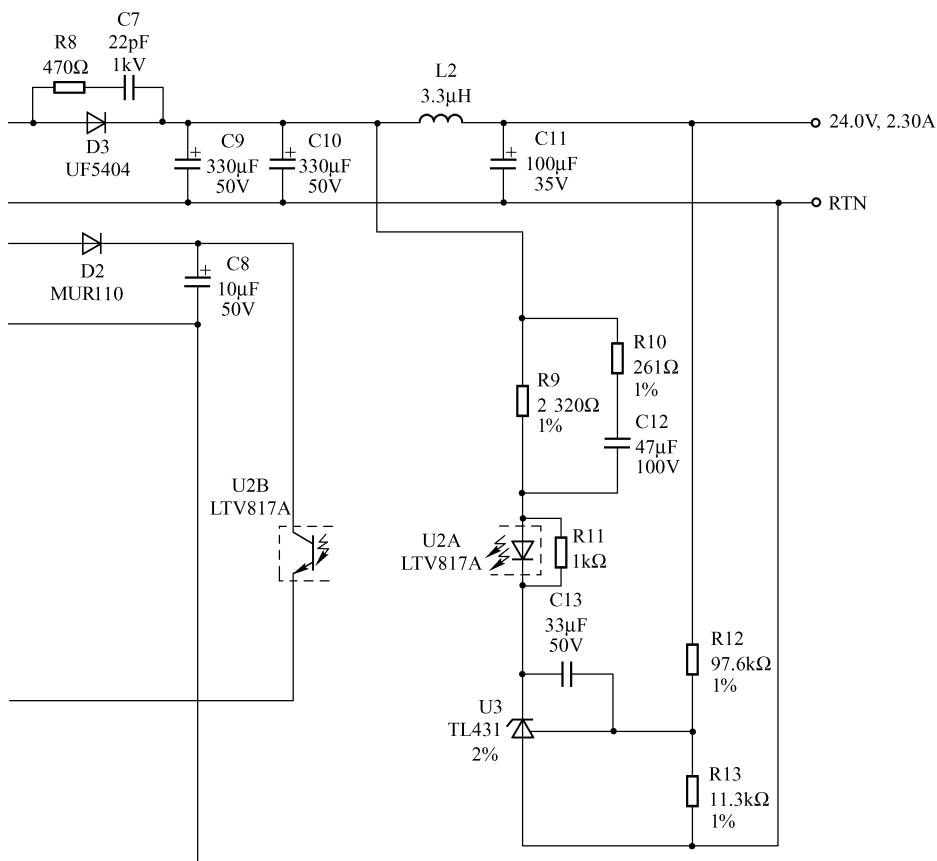


图 2-32 TL431 应用图

整个反馈电路的工作原理是：当输出由于某种原因升高时，分压电压超过 2.5V，TL431

阴极电流上升，光耦电流变大，这时有较强的反馈回到光耦的次级，控制芯片控制的 PWM 脉冲变窄，变压器次级输出的电压变低，回落到 24V；当输出由于某种原因降低时，分压电压低于 2.5V，TL431 阴极电流减小，光耦电流变小，这时有较小的反馈回到光耦的次级，控制芯片控制的 PWM 脉冲变宽，变压器次级输出的电压变高，接近 24V。

2.5.4 开关变压器的设计

变压器用的是铁氧体变压器，型号是 EER28。由图 2-33 可以看出，EER28 的最大功率可达到 56.4W，但这是国外材料的标准，国内许多厂家的磁芯虽然型号也是这个，但参数偏离很大，要仔细挑选供应商。



图 2-33 各种变压器磁芯的参数

在设计结果中，可以看到目前生成的变压器参数，如图 2-34 所示。

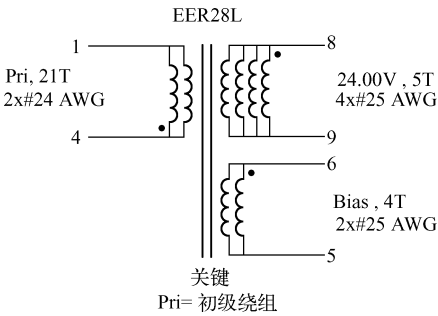


图 2-34 软件自动生成的变压器参数

图 2-34 中的变压器设计得不错，考虑了高频的集肤效应，为了满足输出端达到 2A 的电流，次级用了 4 组绕组并联绕制。但此变压器在工作时初级的脉动较大，会引起吸收回路较大的发热。经过反复实验，调整初级绕组的绕制方法，形成如图 2-35 所示的变压器参数。注意，图 2-34 是在工作频率为 132kHz 的前提下，而图 2-35 则是工作频率为 66kHz 时的参数。初级绕组分为两个组，以三明治的方式绕制可以大大降低变压器的脉动，实际测试表明，吸收回路的发热量相当低。

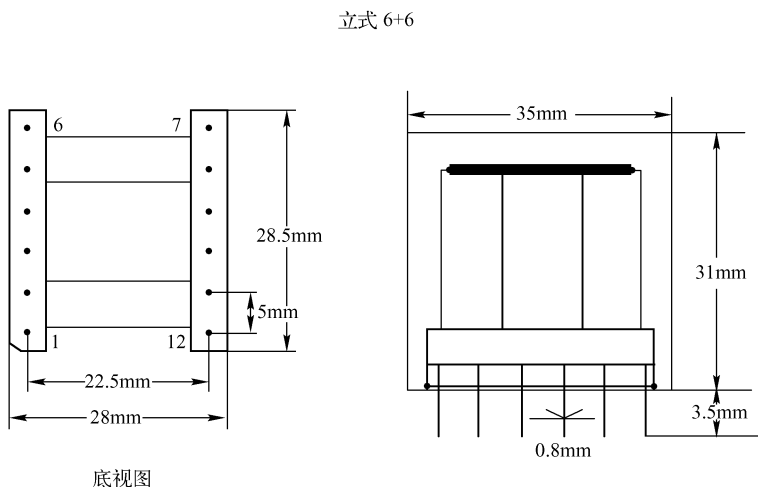


图 2-35 变压器尺寸和脚位图

图 2-35 是变压器尺寸和脚位的初步设计图，在进行 PCB 设计时会用到。它和图 2-36 一起最终确定了变压器 PCB 封装的制作。

图 2-36 是要发给变压器厂家的图纸，另外需要一起提供的图纸还有如图 2-37 所示的绕制结构图，以及绕组说明文件、变压器材料文件和电特性测试规格文件。

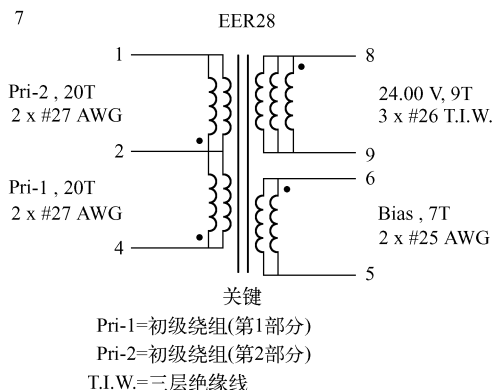


图 2-36 调整后的变压器参数

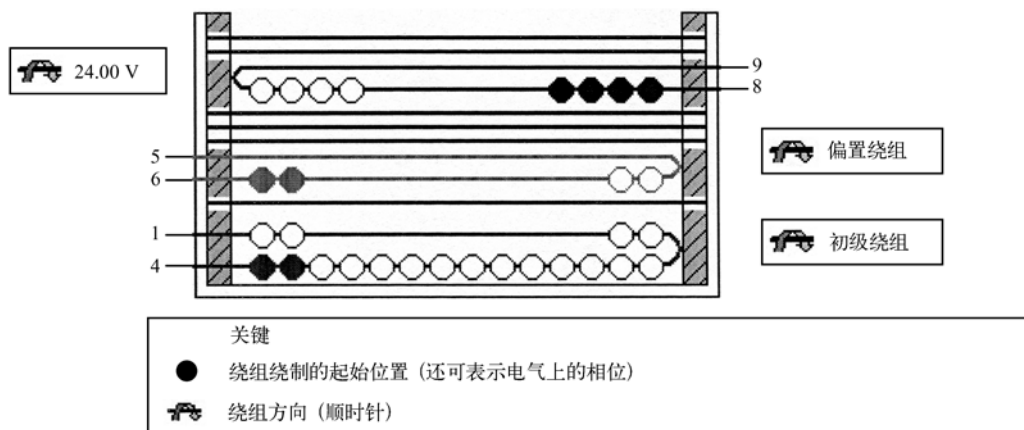


图 2-37 绕制结构图

1. 绕组说明文件

在左侧使用 3.20mm 边距（材料项[3]），在右侧使用 3.20mm 边距（材料项[3]）。

初级绕组：

从引脚 4 开始，使用材料项[6]绕 21 圈（ $\times 2$ 线），在 2 层中从左向右。在第 1 层结束时，继续从右向左绕下一层。在最后一层上，使绕组均匀分布在骨架上。在引脚 1 处结束该绕组。

添加 1 层胶带（材料项[4]）以进行绝缘。

偏置绕组：

从引脚 6 开始，使用材料项[7]绕 4 圈（ $\times 2$ 线）。沿与初级绕组相同的旋转方向进行绕制。使绕组均匀分布在骨架上。在引脚 5 处结束该绕组。

添加 3 层胶带（材料项[4]）以进行绝缘。

次级绕组：

从引脚 8 开始，使用材料项[7]绕 5 圈（ $\times 4$ 线）。使绕组均匀分布在骨架上。沿与初级绕组相同的旋转方向进行绕制。在引脚 9 处结束该绕组。

添加 2 层胶带（材料项[4]）以进行绝缘。

磁芯装配：

装配并固定两半磁芯（材料项[1]）。

浸渍：

在材料项[5]中均匀浸渍，不要采用真空浸渍。

2. 变压器材料

说明如下。

[1]：磁芯，EER28L，NC-2H (Nicera) or Equivalent，开气隙，使 ALG 为 350nH/t。

[2]：骨架，Generic，6 pri. + 6s。

[3]：胶带，聚酯网胶带，宽 3.20mm。

[4]：隔离带，聚酯薄膜（1mil 轴向厚度），宽 22.40mm。

[5]：浸渍。

- [6]: 磁线, 24AWG, 可焊接, 双面涂层。
- [7]: 磁线, 25AWG, 可焊接, 双面涂层。

3. 电特性测试规格文件（见表 2-3）

表 2-3

参 数	条 件	规 格
绝缘强度, VAC	60Hz, 持续 1s, 自引脚 1、2、3、4、5、6 到引脚 8、9	3 000
额定初级电感量, μH	于 $1V_{pk-pk}$ 、典型开关频率、在引脚 1 到引脚 3 之间测量, 此时所有其他绕组均开路	159
容差, $\pm\%$	初级电感量容差	10.0
最大初级漏感, μH	在引脚 1 到引脚 3 之间测量, 此时所有其他绕组均短路	

2.5.5 控制集成电路部分

前面提到过, 为节约成本, 采用 UC3842 的方案, 见图 2-38 和图 2-39。

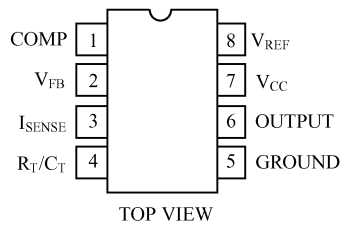


图 2-38 UC3842 外形

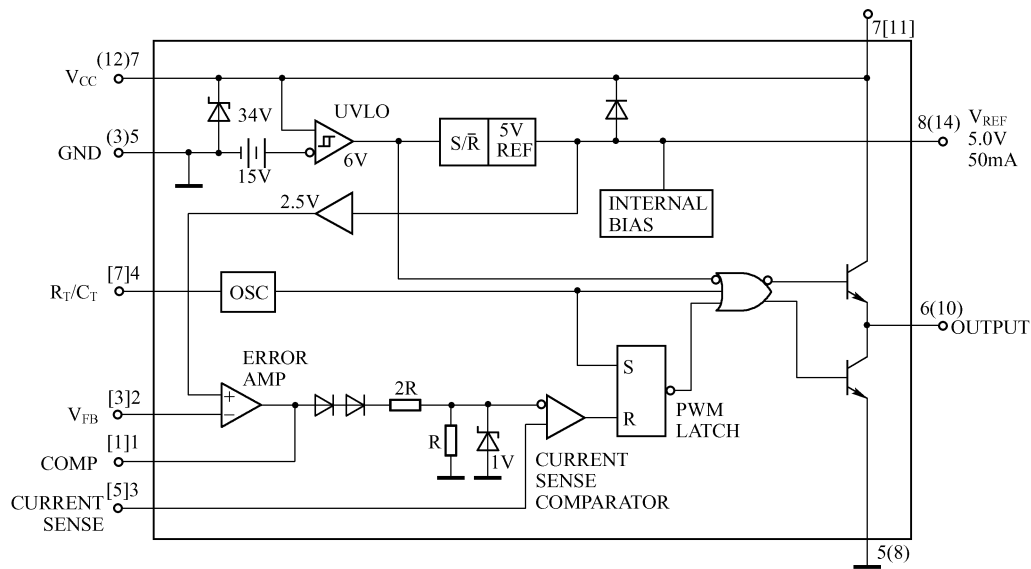


图 2-39 UC3842 内部原理

UC3842 采用固定工作频率脉冲宽度可控调制方式, 共有 8 个引脚, 各引脚功能如

下：①脚是误差放大器的输出端，外接阻容元件，用于改善误差放大器的增益和频率特性；②脚是反馈电压输入端，此脚电压与误差放大器同相端的 2.5V 基准电压进行比较，产生误差电压，从而控制脉冲宽度；③脚为电流检测输入端，当检测电压超过 1V 时缩小脉冲宽度，使电源处于间歇工作状态；④脚为定时端，内部振荡器的工作频率由外接的阻容时间常数决定， $f=1.72/(R_T \times C_T)$ ；⑤脚为公共地端；⑥脚为推挽输出端，内部为图腾柱式，上升、下降时间仅为 50ns，驱动能力为 $\pm 1A$ ；⑦脚是直流电源供电端，具有欠、过压锁定功能，芯片功耗为 15mW；⑧脚为 5V 基准电压输出端，有 50mA 的负载能力。

UC3842 是一种性能优良的电流控制型脉宽调制芯片，该调制器单端输出，能直接驱动双极型的功率管或场效应管。其主要优点是引脚效应小，外围电路简单，电压调整率可达 0.01%，工作频率高达 500kHz，启动电流小于 1mA，正常工作电流为 5mA，并可利用高频变压器实现与电网的隔离。该芯片集成了振荡器、具有高温补偿的高增益误差放大器、电流检测比较器、图腾柱输出电流、输入和基准欠电压锁定电路及 PWM 锁存器电路。

UC3842 是开关电源用电流控制方式的脉宽调制集成电路。与电压控制方式相比，在负载响应和线性调整度等方面有很多优越之处。

该电路主要特点有：

- 内含欠电压锁定电路；
- 低启动电流（典型值为 0.12mA）；
- 稳定的内部基准电压源；
- 大电流推挽输出（驱动电流达 1A）；
- 工作频率可达 500kHz；
- 自动负反馈补偿电路；
- 双脉冲抑制；
- 较强的负载响应特性。

更详细的资料参见 UC3842 datasheet。

在本产品的设计中，UC3842 的连接如图 2-40 所示。

图中，UC3842 采用双闭环接法，内环为电流环，外环为电压环。

首先分析 UC3842 的启动电路。R5 为上电开始时给 UC3842 提供的启动电阻，时间常数为 $0.707 \times R_5 \times C_7 = 1.4s$ ，因此上电后有 1.4s 的延迟。假设辅助绕组由于某种原因没有提供电压给 UC3842，上电后观察到的现象是变压器发出“噗、噗”的声音，然后在输出接一个发光管，可发现发光管突然亮一下，之后灭掉，如此反复。这时由于 UC3842 工作所需的电流比启动的时候来得大，R5 是按照启动设计的，因此无法维持 UC3842 持续的工作电流。当启动后 MOS 管导通，变压器初级有电流，由于变压器初级和辅助绕组的绕向相反，因此与辅助绕组连接的 D8 截止。接着 MOS 管关闭，变压器中的能量要从次级释放。D6 导通，向 C7 充电，进而向 UC3842 供电。UC3842 正常供电电压是 16V 以上，因此将辅助绕组设计成输出电压为 18V 比较合适。R6 和 R27 可防止开关电源进入危险区时自动进入间歇工作状态，起到保护作用。

24V 输出回路的原理和辅助绕组一样，不再重复。

同时，反馈回路去掉了 TL431，使用了 24V 的稳压管。

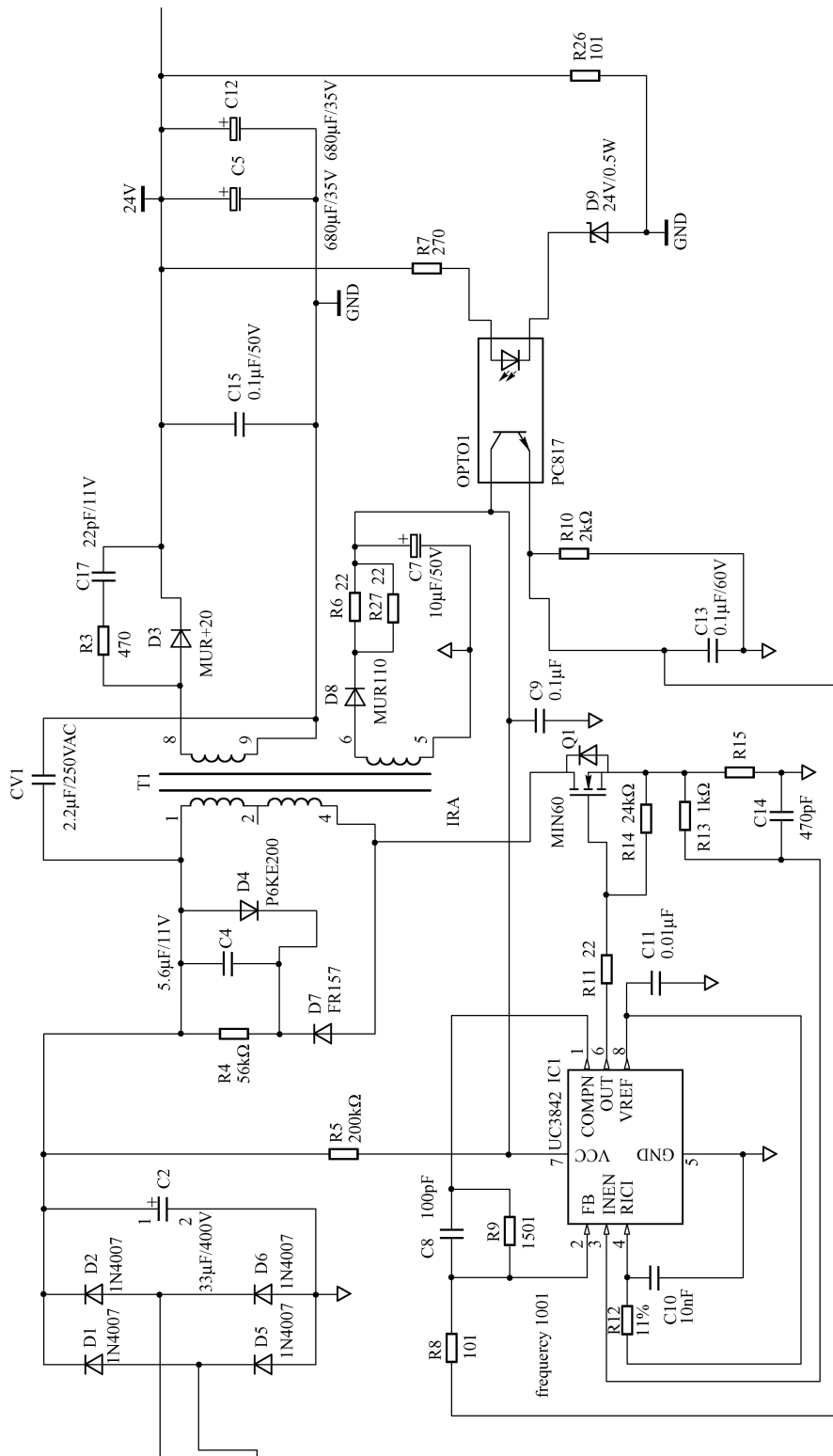


图 2-40 UC3842 开关电源

R26 用于防止开关电源空载，开关电源空载带来的问题是变压器能量没有释放，会引起磁饱和，使开关电源不稳定。

UC3842 的电压反馈工作流程为：当 24V 的输出电压偏高时，PC817 的反馈加强，UC3842 的 2 脚电压升高，由图 2-39 可知，误差放大器正端的电压内部固定为 2.5V，当负端电压升高时误差放大器输出的电压变小，最终导致输出的 6 脚为低电平。C13 是抗干扰用，R10 可增强电源输出的动态响应。R9 是误差放大器的反馈放大电阻，C8 起微分作用，能在输出电压快速动态波动的时候加强反馈。PC817 是非线性光耦，但其特性曲线接近线性，能满足开关电源的反馈要求。

UC3842 的电流反馈工作流程为：R15 为电流取样电阻，R13 和 C14 构成了积分电路，电压进入 UC3842 的电流检测脚 3 引脚。在此用积分电路的意义是过滤 MOS 管上的尖峰电流干扰，防止其对 UC3842 产生影响。当 MOS 管电流变大时，UC3842 的电流误差放大器反转，关闭功率 MOS 管的输出。例如，输出短路的情况下，因为电压环的反馈作用，MOS 电流变大，试图维持输出电压稳定，经过几个周期后，电流越来越大，如果继续增大，则 MOS 管开始烧毁。因此电流反馈及时切断了输出，保护了功率 MOS 管。

R12 和 C10 是定时电路，它和 VREF 连接，VREF 为 5V，定时电路控制了 PWM 的频率，改变电阻电容的数值可使 PWM 的频率不同，以适应诸如 132kHz 和 66kHz 的场合。

图 2-41 是 UC3842 死区时间、频率和电阻电容的关系，在设计开关电源时要仔细考虑。频率越高，变压器体积越小，但材料要求也越高，同时 MOS 功率管的开关损耗大，散热片大；频率越低，变压器体积越大，成本越高，但 MOS 管发热小，散热片小。应该按照综合成本进行设计。

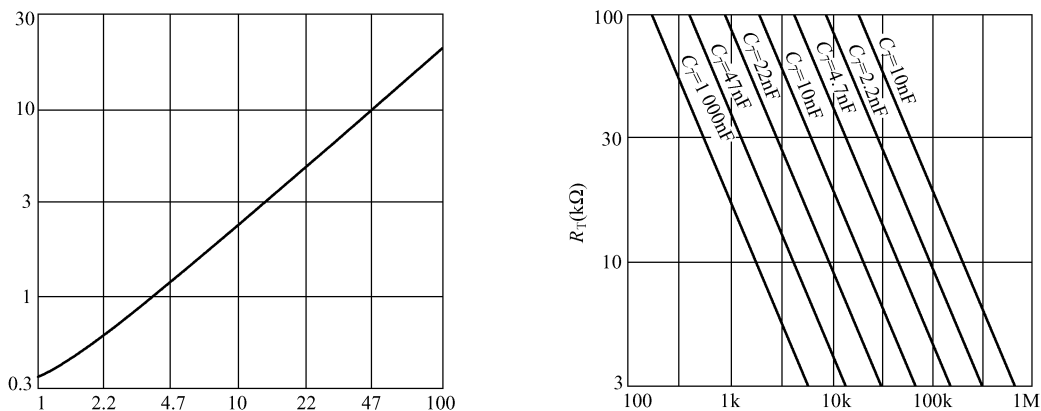
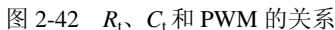


图 2-41 UC3842 死区时间、频率和电阻电容的关系

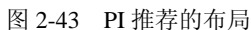
由图 2-42 可知， R_T 大、 C_T 小，频率就低； R_T 小、 C_T 大，频率就高。 R_T 不能太小，否则会发热。



至此，开关电源部分的原理设计全部完成。

开关电源的布线非常讲究，下面先看图 2-43 PI 推荐的布局。

开关电源的布线非常讲究，下面先看图 2-43 PI 推荐的布局。



本设计的 PCB 布局如图 2-44 所示, 变压器下方的电路板上开长条形的槽, 防止高压部

分的干扰对低压部分有影响。CV1 是 Y 安规电容, 不要用一般的高压电容代替。如果不装, 则次级的高频干扰非常大, 共模干扰对单片机构成干扰, 会出现一些莫名奇妙的故障。Y 安规电容分为 Y1 安规电容和 Y2 安规电容, Y1 属于双绝缘 Y 安规电容, 用于跨接一、二次绕组; Y2 则属于基本单绝缘 Y 安规电容, 用于跨接一次绕组对保护大地即 FG 线。安规电容是指用于以下场合的电容器: 电容器失效后, 不会导致电击, 不危及人身安全。特别指出: 作为安全电容的 Y 电容, 要求必须取得安全检测机构的认证。Y 电容外观多为橙色或蓝色, 一般都标有安全认证标志(如 UL、CSA 等)和耐压 AC 250V 或 AC 275V 字样。然而, 其真正的直流耐压高达 5 000V 以上。

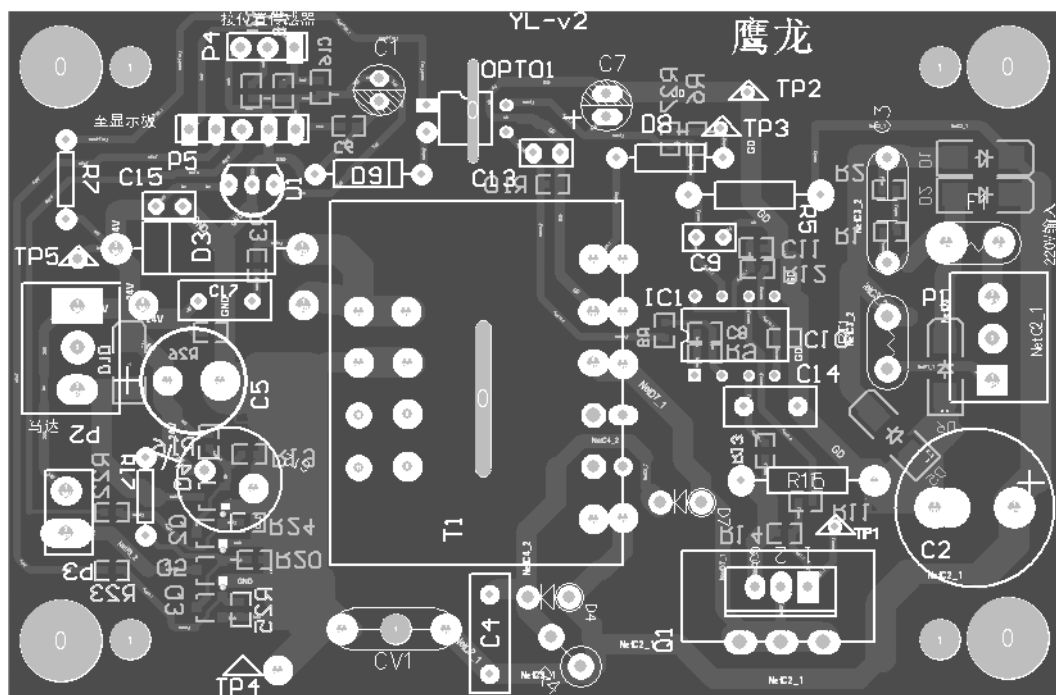


图 2-44 足疗机电源板的 PCB 布局

PCB 上要求焊盘尽量大, 线尽量粗, 开关管这样的强干扰器件要远离 UC3842 对干扰敏感的电路。变压器初级和次级之间的 PCB 要开槽, 以减小通过 PCB 材料传导过来的感应电流。

2.6 控制器其他部分设计

图 2-45 是 5V 的稳压电路, C1 和 C6 分别对低频和高频进行滤波。由于单片机运行时电流很小, 算上 8 个发光管每个 5mA 电流, 最多同时 4 个点亮, 一共为 20mA, 然后是蜂鸣器, 瞬时工作, 可以忽略, 再加上红外槽形光耦, 大约在 10mA 内, 总共 30mA 足够。这样, 78L05 的功耗为 $(24V - 5V) \times 0.03A = 0.57W$, 可以满足要求。

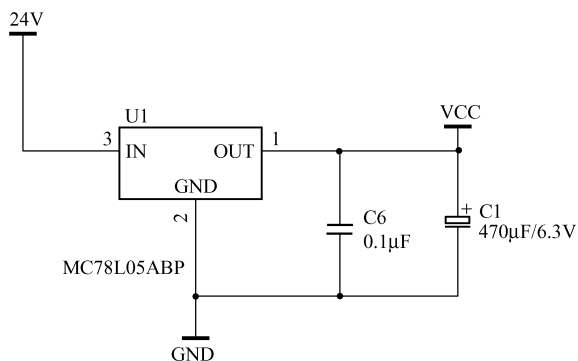


图 2-45 78L05 稳压电路

槽形光耦的封装如图 2-7 所示，其内部原理很简单，就是一个红外的发射管和一个红外的接收管。槽形光耦的接法如图 2-46 右侧所示。

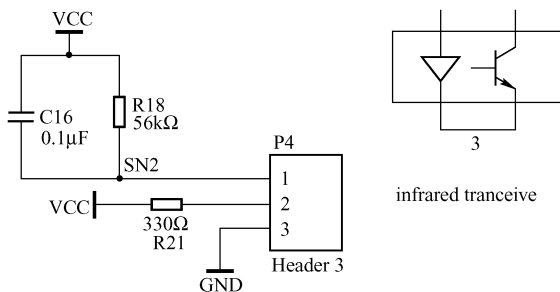


图 2-46 定位器用槽形光耦电路

图 2-47 是实际的 PCB，和图 2-46 右侧对应。发射管的阳极接到 P4 的 2 脚，阴极接到 P4 的 3 脚，因此，红外发射管是一直开通的。而接收管在槽中没有遮挡物时是导通的，因此 P4 的 1 脚为低电平；当有遮挡板时，接收管不导通，P4 的 1 脚为高。单片机检测电平状态可知道目前是否有遮挡物进入，也就是是否到定位点了。

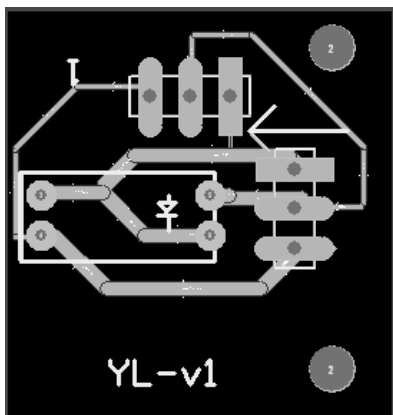


图 2-47 槽形光耦的接法

图 2-48 是加热驱动电路，通过控制 PWM 的脉宽调整电热丝的电压。也许有人会问为什么不功率管处于放大状态。这就是功率管的开关和放大状态的区别。驱动扬声器这类设备应该让功率管工作在放大状态，而驱动电热丝、电动机这类负载只能处于开关状态。因为功率管本身的功耗是功率管上电压降和电流的乘积，如果电流为 1A，电压降为 10V，那就是 10W，对于贴片式 SOT-23 封装的功率管几秒后就会烧毁。而开关状态下的功率管饱和压降很低，如 0.2V 的压降，2A 的电流，功耗只有 0.4W。图 2-48 中 Q5 前加了 9014 NPN 管，为的是和 5V 的系统接口。R17 和 R20 是分压电阻，使 Q5 的栅极电压满足工作要求，如果不加会超过额定栅极电压而被击穿。R23 是驱动 9014 的限流电阻，R25 是个防止布线干扰的。如果没有 R25，则从单片机到 Q3 基极的引线可能受外界干扰后引起误动作。R25 可吸收这些干扰，布线时要让它和 Q3 非常接近。由于电热丝是电阻负载，因此不需要续流管。

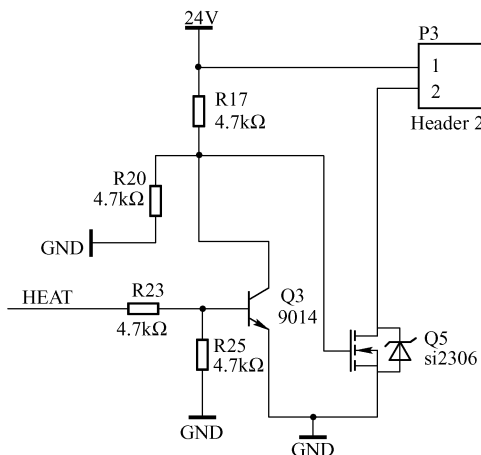


图 2-48 加热驱动电路

图 2-49 是电动机驱动电路，与加热驱动电路基本一样，只是多了一个续流管 D10。

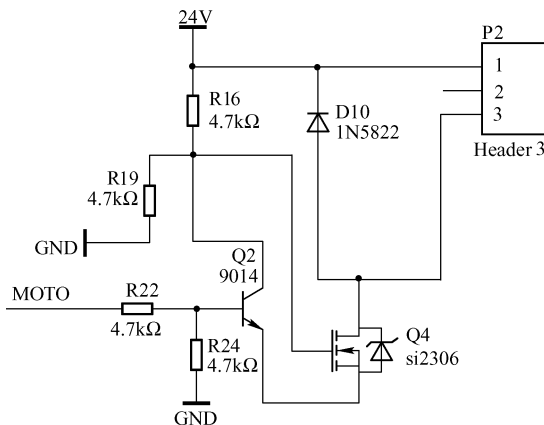


图 2-49 电动机驱动电路

由于 PWM 的频率很高，并且电流比较大，续流管要求采用肖特基管。如果不装 D10，Q4 断开时电动机的电流引起很高的感生电压，会将 MOS 管击穿。有了续流管，在断开的瞬间，Q4 的漏极电压接近 24V。

驱动电路和开关电源布线在同一个 PCB 上，见图 2-44。

2.7 给 PCB 代工厂提交的资料

图 2-50 和图 2-51 是准备好提交给焊接工厂的资料，包括定位板和电源板的要求。

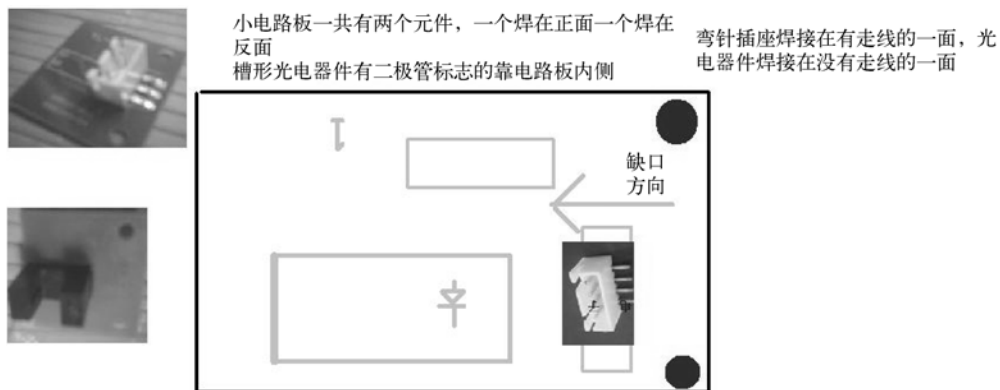


图 2-50 定位板焊接要求

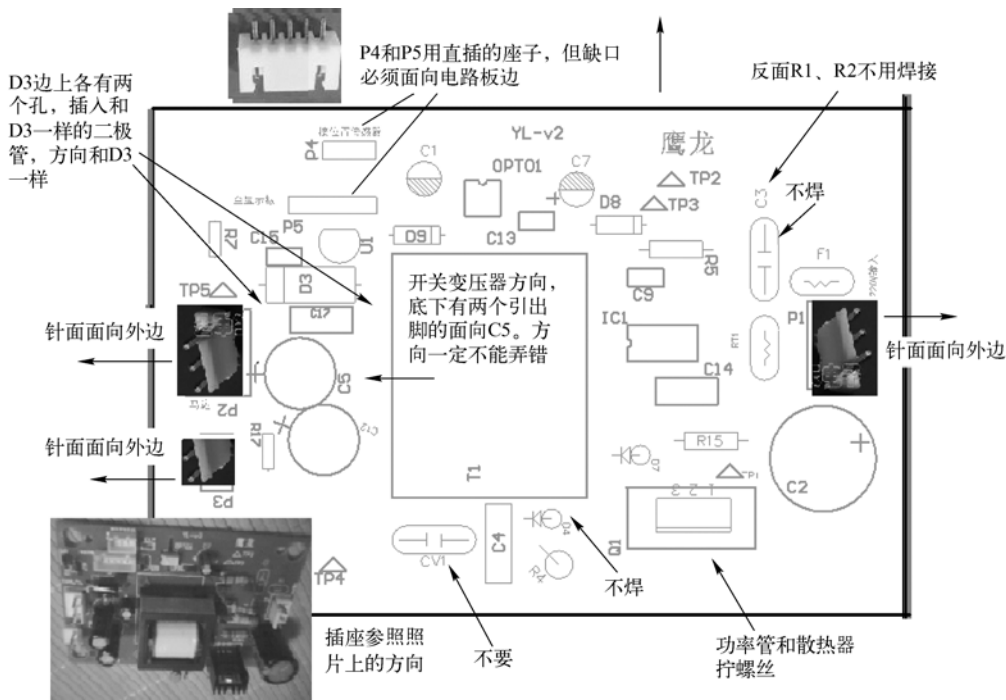


图 2-51 电源板焊接要求

电源板采用的是单面板，既有直插件也有贴片元件。所有直插元件插在正面，贴片元件焊接在反面，反面是走线层。

电源板焊接清单见表 2-4。

表 2-4 电源板焊接清单

名 称	标 号	备 注	数量
9014 三极管	Q2, Q3	SOT-23	2
0.01μF 电容	C11	0603	1
0.1μF 电容	C6, C16	0603	2
0.1μF/50V 电容	C13, C15	直插	2
100pF 电容	C8	0603	1
10nF 电容	C10	0603	1
2.2nF/250V AC 安规电容	CV1	直插	1
22pF/1kV 电容	C17	直插	1
33μF/400V 电容	C2	RB.3/.6'直插	1
100pF 电容	C14	rad-0.2	1
5.6nF/1kV 电容	C4	rad-0.3	1
24V 稳压二极管	D9	直插	1
P6KE200 瞬变吸收二极管	D4	直插	1
1N4007 整流二极管	D1, D2, D5, D6	贴片	4
ss34 肖特基管	D10	贴片	1
FR157 快恢复管	D7	直插	1
MUR420 超快恢复管	D3	直插	1
MUR110 超快恢复管	D8	直插	1
10μF/50V 电解电容	C7	直插	1
470μF/6.3V 电解电容	C1	直插	1
680μF/35V 电解电容	C5, C12	RB.2/.4'直插	2
2A 熔丝	F1	直插	1
Header 2, 2-Pin	P3	3.96mm 插座	1
Header 3, 3-Pin	P1, P2	3.96mm 插座	2
Header 3, 3-Pin	P4	白色接插件 XH2.54, 间距 2.54mm	1
Header 5, 5-Pin	P5	白色接插件 XH2.54, 间距 2.54mm	1
si2306, HEXFET Power MOSFET	Q4, Q5	SOT-23	2
78L05 三端稳压	U1	TO-92	1
PC817 光耦	OPTO1	DIP4 直插	1
4N60, N-channel MOSFET	Q1	TO-220	1
0.5Ω/2W 电阻	R15	直插	1

续表

名 称	标 号	备 注	数量
10kΩ电阻	R8,R26	0603	1
150kΩ电阻	R9	0603	1
200kΩ/1W 电阻	R5	直插	1
22Ω电阻	R6, R11,R27	0603	3
2kΩ电阻	R12,R10	0603	2
5ΩNTC 热电阻	RT1	直插	1
56kΩ电阻	R18	0603	1
270 电阻	R7	axial-0.3	1
56kΩ/2W 电阻	R4	直插	1
330Ω电阻	R21,R3	0603	2
4.7kΩ电阻	R16, R19, R20, R22, R23, R24, R25	0603	7
4.7kΩ电阻	R17	直插	1
1kΩ电阻	R13	0603	2
TRA 开关变压器	T1	定做	1
UC3842 集成电路	IC1	DIP8	1
24kΩ电阻	R14	0603	1
散热器	YA25, 25×23×16 (长×宽×高), 单孔		1

焊点的计算：用来估算电源板贴片加工的费用。

电源板：接插件元件，16 个点；

2 个脚的元件， $22 \times 2 = 44$ 点；

3 个脚的元件， $2 \times 3 = 6$ 点；

4 个脚的元件， $1 \times 4 = 4$ 点；

8 个脚的元件， 1×8 点；

变压器元件，10 点；

散热器元件，1 点。

直插一共 89 个点。

贴片：2 个脚的， $30 \times 2 = 60$ 点；

3 个脚的， $4 \times 3 = 12$ 点。

电源板贴片一共 72 个点。

按照贴片厂的报价， $89 \times 0.025 + 72 \times 0.015 = 3.3$ 元。这个价格要看谈判技巧，如果没有持续的批量，一般代工厂不愿意做。

好了，到目前为止硬件设计和生产流程全部介绍完毕。硬件设计成果见图 2-52，从左到右依次是定位板、CPU 板、电源和电动机驱动板。



图 2-52 硬件设计成果

2.8 足疗机软件设计

下面介绍足疗机的软件设计。按照前面的需求分析，总结功能如下：

- 红外接收与解码；
- 键盘处理和 LED 的显示；
- 电动机两种工作模式；
- 电动机三挡速度的设置；
- 电热丝高低温度的控制；
- 定位程序。

表 2-5 和表 2-6 分别是程序用到的变量名和函数名。

表 2-5 程序用到的变量名

变 量 名	类 型	功 能
pgm_style	字符	电动机模式变量 0：停止；1：间断模式；2：持续模式
m_tmp	字符	电动机控制 PWM 时间变量
btm_tmp	字符	电热丝 PWM 时间变量
heat_var	字符	加热判断 0：不加热；1：低温；2：高温
fast_slow_var	字符	电动机速度挡 0：低速；1：中速；2：高速
HOT_Hi	比特	高温指示灯逻辑判断变量 0：亮；1：灭
HOT_Lo	比特	低温指示灯逻辑判断变量 0：亮；1：灭
run_tag	比特	机器运行变量 0：停；1：运行

续表

变 量 名	类 型	功 能
PGM_I	比特	间断模式 LED 灯 1: 亮; 0: 灭
PGM_II	比特	持续模式 LED 灯 1: 亮; 0: 灭
FAST_SLOW_I	比特	低速指示灯 1: 亮; 0: 灭
FAST_SLOW_II	比特	中速指示灯 1: 亮; 0: 灭
FAST_SLOW_III	比特	高速指示灯 1: 亮; 0: 灭
speed	整型变量	速度的 PWM 变量
cnt	字符变量	软件模拟计数器
SN2	I/O 端口 sbit	槽形光耦输入 1: 有挡板进入; 0: 无挡板
S1	I/O 端口 sbit	停止按钮 0: 按下; 1: 未按下
S2	I/O 端口 sbit	加热按钮 0: 按下; 1: 未按下
S3	I/O 端口 sbit	模式按钮 0: 按下; 1: 未按下
S4	I/O 端口 sbit	速度按钮 0: 按下; 1: 未按下
infrared	I/O 端口 sbit	红外遥控输入 0: 信号低电平; 1: 信号高电平
need_position	比特	电动机是否需要定位 1: 需要; 0: 不需要
virtual_key	遥控器按键	分别对应面板上的 S1、S2、S3、S4

表 2-6 程序用到的函数名

函 数 名	功 能
INT0_Routine	红外遥控
time1	PWM 产生, 改变电压
displ	扫描式显示
Keyscan	键盘处理
running_mode_1	间断运行模式
position	电动机输出轴开始位置定位函数

2.8.1 红外通信设计

首先谈谈单片机的资源分配问题。对时间要求高的是红外和 PWM，其余的直接在主程序中处理。红外接收用外中断 0 来实现，由于采用的单片机没有 PWM 功能，为了能够控制电动机和电热丝的电压，用定时器模拟 PWM 的产生。红外通信用最高级的中断，而两个 PWM 分别控制电动机和电热丝，在处理红外通信时 PWM 会产生非常短暂的停顿，由于红外通信的时间是在 200ms 内，如此短的时间对电动机输出不会产生明显的影响，因此也就

是不影响用户的使用体验。

以下为红外通信中断程序，其输出是按键号，就是对应停止、模式、速度、高低温中的一个。从图 2-10 主控板原理图可以看出，红外接收电路 VS1838B 的输出连接到 STC11F02 的外部中断 0，当没有红外信号进来时，引脚为高电平，只要有红外信号，便成为低电平，从而引起外中断程序执行。

图 2-53 是遥控器发出后接收端经过 VS1838B 检波输出的波形。注意，这和 VS1838B 发出的原始波形是不同的，原始波形是 38kHz 的经过调制后的载波，而 VS1838B 内部已经对载波进行处理，输出的是调制波。如果不用 VS1838B 而用分立元件自己做，需要注意这个问题。

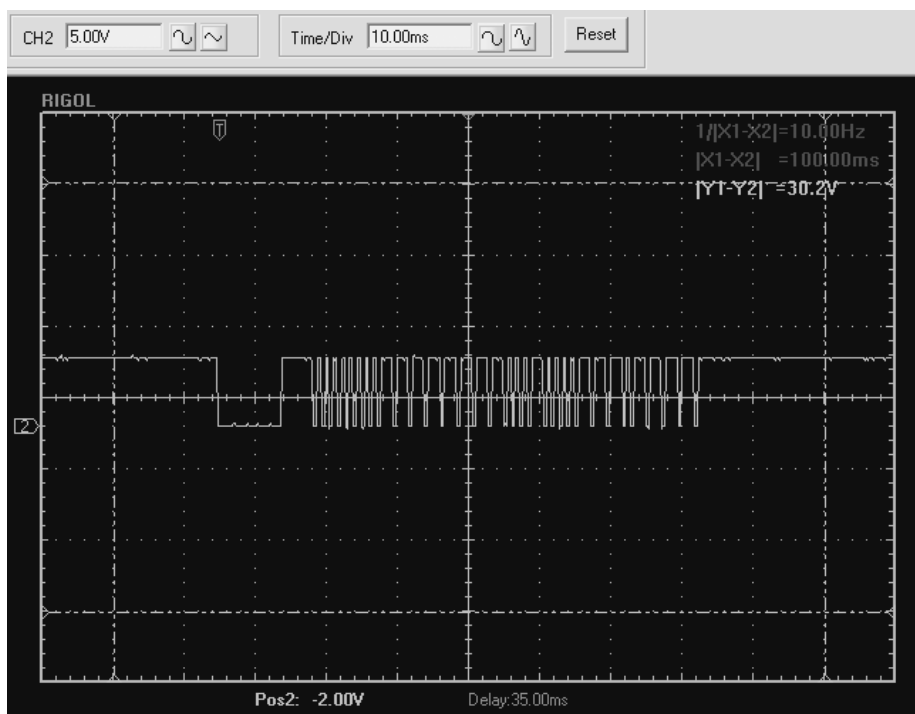


图 2-53 红外遥控波形

从波形上可以看到，开始 9ms 为低，之后 4.5ms 为高，表示这是红外数据的开始，然后便是数据通信。由于要计算红外高低电平长短，因此还需要定时器 0 的配合。

以下是相应的红外接收程序。

```
void INT0_Routine(void) interrupt 0 using 1
{
    char i,j;

    TH0=0;
    TL0=0;
    while(infrared!=1) displ();
```

```

if(TH0>12)
    { TH0=0;
      TL0=0;}
else return;

while(infrared!=0) displ();
if(TH0>6)
    { TH0=0;
      TL0=0;}
else return;

for(i=0;i<4;i++)
{
    mdata[i]=0;
    for(j=0;j<8;j++)
    {

        while(infrared!=1) displ();
        TH0=0;TL0=0;
        while(infrared!=0) displ();

        if(TH0>1)
            mdata[i]=0x1<<j;
    }
}

if(mdata[0]!=0) return;
if((mdata[1]&0xff)!=0xff) return;

if((mdata[2]==0)&&(mdata[3]==0xff)) virtual_key=1; //S1
if((mdata[2]==4)&&(mdata[3]==0xfb)) virtual_key=2; //S2
if((mdata[2]==8)&&(mdata[3]==0xf7)) virtual_key=3; //S3
if((mdata[2]==0xc)&&(mdata[3]==0xf3)) virtual_key=4; //S4
}

```

红外数据的格式分为 4 字节，第一字节定义为 0，第二字节定义为 0xff，第三字节定义为数据 A，第 4 字节为数据 A 的反码。

定义键 1 为 0，键 2 为 4，键 3 为 8，键 4 为 0x0c。

以下分析代码。首先，这个中断是低电平引起后才进入的。

```
void INT0_Routine(void) interrupt 0 using 1
```

定义中断 0，告诉编译器使用寄存器组 1。

```
char i,j;
```

定义循环变量。

```
TH0=0;
TL0=0;
```

清零定时器 0。注意，主程序在初始化时就让定时器 0 一直运行着。

```
while(infrared!=1) disp1();
```

等待低电平结束，因为显示程序是主程序中循环调用的，并未用中断实现，因此这里如果一直等待显示会停下来，表现的结果是 LED 灯在这段时间一直亮。为了避免这种情况出现，需要调用显示程序。

```
if(TH0>12)
{ TH0=0;
  TL0=0;}
else return;
```

如果 $TH0>12$ ，说明低电平持续时间至少为 9ms，否则是干扰信号，忽略并退出。如果是正常，就清零定时器 0。

```
while(infrared!=0) disp1();
if(TH0>6)
{ TH0=0;
  TL0=0;}
else return;
```

这一段和上一段意思一样，不同的是持续时间为 $TH>6$ ，也就是 4.5ms 高电平时间。

```
for(i=0;i<4;i++)
```

接收 4 字节，因此循环 4 次。

以下介绍对单独 1 字节的接收。

```
mdata[i]=0;
```

首先是对当前接收的字节清零，以消除上次接收的数据。

```
for(j=0;j<8;j++)
```

然后对每一比特进行处理，也就是高低电平的处理，1 字节共 8 比特，如图 2-54 所示。

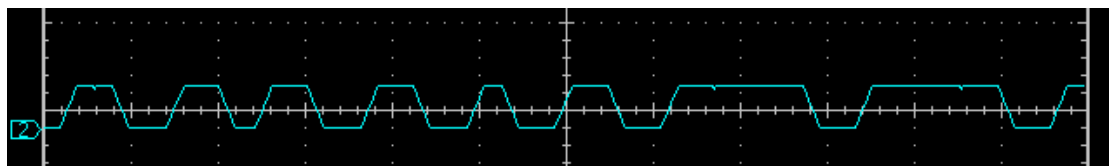


图 2-54 红外遥控中 0 和 1 的波形

对比特进行分析可以看出，1 和 0 的不同在于高电平持续的时间不同，而在低电平时间上是一样的。

以下开始进行比特的接收。

```
while(infrared!=1) displ();
```

等待高电平的到来，在等待的同时调用 LED 显示程序，避免 LED 停止扫描显示。

```
TH0=0;TL0=0;
```

高电平已经来了，立刻清零定时器 0。

```
while(infrared!=0) displ();
```

等待高电平结束，在等待的同时调用 LED 显示程序，避免 LED 停止扫描显示。

```
if(TH0>1)
mdata[i]=0x1<<j;
```

如果 TH0>1，则一定是持续时间满足了 1 的高电平时间，因此字节的对应位置位；否则就是 0 的高电平时间，对应位不用做任何动作，因为前面字节初始化时就是 0 了。

```
if(mdata[0]!=0) return;
if((mdata[1]&0xff)!=0xff) return;
```

由于遥控器的红外发射通信定义了前面 2 字节固定是 0 和 0xff，因此如果接收到的不是这个，则肯定是出问题了，要么用的遥控器有问题，要么受到干扰。此时应该忽略本次接收，退出中断。

```
if((mdata[2]==0)&&(mdata[3]==0xff)) virtual_key=1; //S1
if((mdata[2]==4)&&(mdata[3]==0xfb)) virtual_key=2; //S2
if((mdata[2]==8)&&(mdata[3]==0xf7)) virtual_key=3; //S3
if((mdata[2]==0xc)&&(mdata[3]==0xf3)) virtual_key=4; //S4
```

以上 4 句代码的含义相同，以第一句为例，如果第三字节是 0 并且第四字节是 0 的反码 0xff，则对应的红外按键是 1，也就是和控制面板上按 S1 按键的效果相同。在 4 字节全部接收完毕后，中断程序结束。输出结果是 virtual_key。

2.8.2 PWM 产生

下面介绍 PWM 模拟产生程序，该程序采用定时器 1 中断模拟产生。

在程序的定义部分有：

```
sbit MOTO=P1^5;
sbit HEAT=P1^6;
```

表示将电动机 PWM 输出 MOTO 定义为 P1.5，将加热 PWM 输出 HEAT 定义为 P1.6。

```
void time1(void) interrupt 3 using 1 //产生 10ms 定时中断
{
    static unsigned char m_tmp=0;
    static unsigned char btm_tmp=0;
```

首先定义电动机控制 PWM 时间变量 `m_tmp` 和电热丝 PWM 时间变量 `btm_tmp` 为静态变量，也就是每次重新载入时能保持上次的数据。

```
if(m_tmp<=(MAX_VAL-speed)) MOTO=1;
else MOTO=0;
```

上面代码中 `MAX_VAL` 是 PWM 的周期，`speed` 是当前电动机速度的 PWM 占空时间，从足疗机电源板原理图可以看到，PWM 到电动机的实际驱动是反相的，也就是当 `MOTO` 为高电平时电动机不转，为低电平时才转动。由图 2-55 可以看出，白色部分是控制电动机开通的时间，右边灰色部分是控制电动机断电的时间。从代码可以知道，当目前时间进入右边区域时，控制电动机断电。

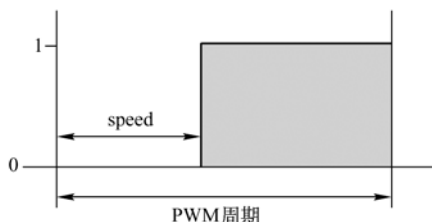


图 2-55 PWM 周期

```
if(btm_tmp<=(MAX_VAL-hot)) HEAT=1;
else HEAT=0;
```

上述两句代码的意思和电动机控制的含义相同，是用来控制电热丝的。

```
if(btm_tmp++>=MAX_VAL) btm_tmp=0;
```

如果 PWM 时间变量 `btm_tmp` 达到周期时间，则清零该变量。

```
if(m_tmp++>=MAX_VAL) m_tmp=0;
```

如果 PWM 时间变量 `m_tmp` 达到周期时间，则清零该变量。

```
displ(); //刷新显示，防止显示停止

}
```

2.8.3 定时扫描显示

LED 的扫描显示和普通静态显示相比的优点是可以节约 I/O 的数量，也就是简化硬件，当然缺点是软件编写麻烦。对于家用电器这类对成本比较敏感的产品，可以采用这种方法。为方便学习，以下将显示的原理图用图 2-10 重新绘出，如图 2-56 所示。原理很简单，就是 LED 灯分时显示，依靠视觉暂留现象，使 LED 看上去的效果和单独的 I/O 点亮一样。下面结合显示子程序软件进行说明。

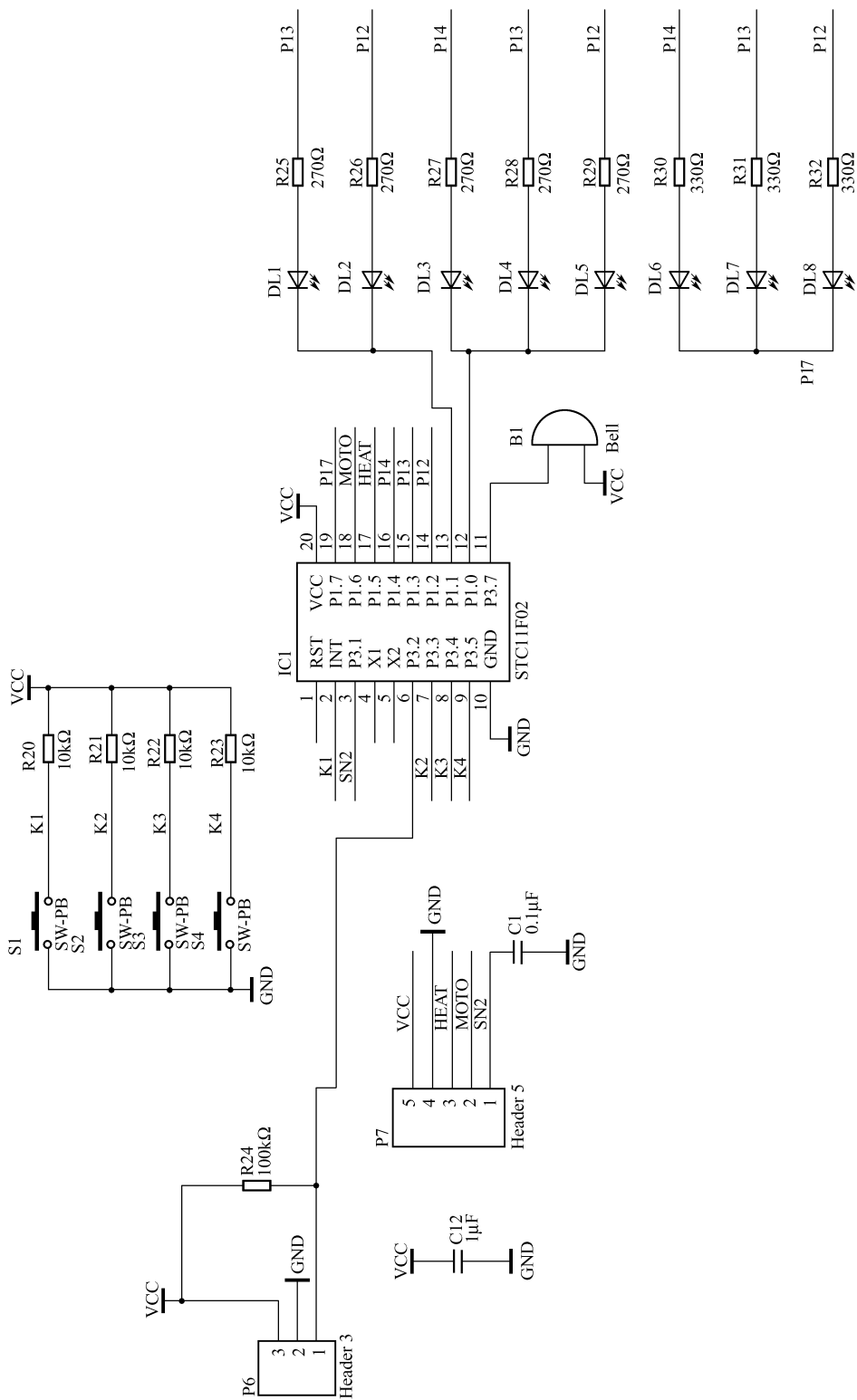


图 2-56 LED 和按键

以上灯和功能的对应为：

- DL1——启/停；
- DL2——低温；
- DL3——高温；
- DL4——模式 1；
- DL5——模式 2；
- DL6——低速；
- DL7——中速；
- DL8——高速。

键盘和功能的对应为：

- S1——停止；
- S2——加热；
- S3——模式；
- S4——速度。

```
void displ()
{
    cnt++;
```

变量 cnt 表示虚拟计数器。

```
if(cnt==40) cnt=0;
```

如果 cnt 到了显示的循环周期，则清零。

```
switch(cnt){
```

按照 cnt 目前的数值进行切换显示。

```
case 10:
    clr();
```

如果目前 cnt 到 10 了，则关闭所有的 LED 灯。

```
P1_7=0;
```

选通图 2-56 最下面三个 LED 的阴极。

```
P1_2=FAST_SLOW_III?1:0;
P1_3=FAST_SLOW_II?1:0;
P1_4=FAST_SLOW_I?1:0;
break;
```

根据目前的速度挡次变量来决定对应的 LED 是否点亮。例如，第一句表示，如果 FAST_SLOW_III 变量为真，则选 1 即 P1_2 为高，反之为低。

```
case 20:
```

```
clr();
```

如果目前 cnt 到 20 了，则关闭所有的 LED 灯。

```
P1_0=0;
```

选通图 2-56 中间三个 LED 的阴极。

```
P1_2=PGM_II?1:0;
P1_3=PGM_I?1:0;
```

根据目前的模式选择变量来决定对应的 LED 是否点亮。例如，第一句表示，如果 PGM_II 变量为真，则选 1 即 P1_2 为高，反之为低。

```
P1_4=HOT_Hi?1:0;
```

根据目前的高温选择变量 HOT_Hi 来决定对应的 LED 是否点亮。如果 HOT_Hi 变量为真，则选 1 即 P1_4 为高，反之为低。

```
break;

case 30:
clr();
```

如果目前 cnt 到 30 了，则关闭所有的 LED 灯。

```
P1_1=0;
```

选通图 2-56 最上方两个 LED 的阴极。

```
P1_2=HOT_Lo?1:0;
```

根据目前的低温选择变量 HOT_Lo 来决定对应的 LED 是否点亮。

```
if(run_tag==1) P1_3=1;
else {
    if(tm++>500) P1_3=1;
    else P1_3=need_position;
}
break;
```

上面代码的第一句表示如果机器运行标志置位，则对应的运行灯点亮。

下面几句的意思是，当机器没有启动时，灯表现为一闪一闪，如果机器正在定位，对应的灯也为闪烁。当然这需要结合整体的程序才能完全理解。一般而言，开发产品做程序时先编写主要功能，并进行调试，一些次要功能可最后添加。

```
}
}
```

2.8.4 键盘处理程序

下面的程序是以查询方式执行的，可以发现里面用了巧妙的方法避免查询引起的延迟。

```
char keyscan()
{
    unsigned char i;
```

定义循环变量 i。

```
if(virtual_key!=0){
```

如果红外遥控没有按键按下，就进入面板上键盘按下与否的判断程序。

```
if((P3&0x39)!=0x39)    {if(keytag==1) return -1;}
```

通过判断 K1、K2、K3、K4 是否都是高电平来了解是否有按键按下。如果是，则进一步判断 keytag 标志是否为 1，这是为了防止持续按一个按键重复执行动作。Keytag 表示当一个按键按下后置位，松开后清零。如果是持续按的，就退出，不再执行下面的程序。

```
else    {keytag=0;return -1;}
```

如果按键都是松开的，则清零 keytag 标志，并返回。

```
for(i=0;i<250;i++);
for(i=0;i<250;i++);
```

延迟时间 20ms，应对键盘的弹簧片抖动。

```
if((P3&0x39) == 0x39)
    {keytag=0;return -1;}
}
```

延迟大约 20ms 后，如果发现键都松开了，说明是干扰导致的，不是真正地按下键。这种情况在键盘线比较长或者干扰大的场合经常发生，为了产品的稳定性，在软件上需要进行抗干扰设计。

```
keytag=1;
```

如果确实是按下键，置键盘按下标志。

```
if(keytag==1||virtual_key!=0) {
```

如果目前面板上的键或者遥控器上的键有按下，继续执行。

```
if(S1==0||virtual_key==1) {
```

如果按下了 S1 按键，或者在遥控器上按了停止按钮，执行以下程序。

```
if(pgm_style!=0) need_position=1;
```

pgm_style==0 表示停止, pgm_style==1 表示模式 1, pgm_style==2 表示模式 2, 如果目前不是停止状态, 则一定是运行状态。从运行状态进入到停止状态需要让电动机运转到定位处, 这样才可以把脚从按摩器中取出。

```
run_tag=0; pgm_style=0; heat_var=0;
```

运行标志 run_tag 清零, pgm_style 清零, 加热变量 heat_var 清零, 此为停止的状态。

```
if(S2==0||virtual_key==2) {heat_var=(heat_var+1)%3; }
```

如果按了 S2 按键, 或者遥控器上的加热键按下过, 则加热变量增加 1, 0 表示不加热, 1 表示低温, 2 表示高温。

```
if(S3==0||virtual_key==3) {pgm_style=(pgm_style+1)%3;
```

如果按了 S3 按键, 或者遥控器上的模式按钮按下过, 则模式加 1, 0 表示停止, 1 表示模式 1, 2 表示模式 2。

```
if(pgm_style==0) need_position=1;}
```

如果刚刚开机, 则进行定位。

```
if(S4==0||virtual_key==4) fast_slow_var=(fast_slow_var+1)%3;
```

如果按了 S4 按键, 或者遥控器上的速度按钮按下过, 则速度变量加 1, 0 表示慢速, 1 表示中速, 2 表示快速。

```
}  
bell();
```

蜂鸣器叫一下, 提示用户。

```
virtual_key=0;
```

清零遥控按键标志。

```
}
```

2.8.5 间断运行模式程序

注意, 以下程序为了理解方便, 对原始程序进行了适当的删减, 主要是涉及延迟的事项。程序的按钮为查询方式, 所以如果执行以下程序时按下按钮, 将无法响应, 因此在原始程序的延迟中添加了对按钮的查询, 并且一旦发现按钮按下, 则立刻退出并执行按钮程序。由于查询按钮用时很短, 因此不会对延迟产生大的影响。

```
int running_mode_1(){  
char i,j,k;
```

定义循环变量。

```
unsigned int speed_bak;
speed_bak=speed;
```

速度有两个变量，`speed` 表示当前设置的速度，`speed_bak` 是对原来的速度进行备份。

```
for(k=0;k<10;k++){
```

以下节拍循环运行 10 次。

```
for(j=0;j<6;j++){
```

内循环对应断续模式节拍表（见表 2-1）的第 3 列。

```
for(i=0;i<7;i++){
```

对应断续模式节拍表的 1~11 项的(1,2)、(3,4)、(5,6)、(7,8)、(9,10)、(11,12)，每一对的执行都相同，因此循环 6 次重复执行。

```
speed=speed_bak;
```

`speed` 恢复初始设置，也就是正常运行的速度。

```
delay(67);
```

按照表 2-1 中序号 1 的要求延迟。

```
speed=0;
```

按照表 2-1 中序号 1 的要求，速度设置为 0。

```
delay(135);
```

按照表 2-1 中序号 1 的要求延迟。

```
    }
    for(i=0;i<13;i++)
    {
```

按照表 2-1 中序号 2 的要求循环 13 次。

```
speed=speed_bak;
```

速度恢复为初始值。

```
delay(30);
speed=0;
```

按照表 2-1 中序号 2 的要求，速度设置为 0。

```
delay(70);
```

按照表 2-1 中序号 2 的要求延迟。

```
    }
  }
  for(i=0;i<9; i++){
```

执行表 2-1 中序号 3~11。

```
    speed=0;
    longdelay(tab[i*2]);
```

延迟需要查表，因为延迟节拍的要求都放在延迟表格中，在程序的开头有相关全局定义如下：

```
const unsigned char  tab[18]={30,80,30,80,30,80,30,160,30,70,10,20,10,20,30,10,30,0};
    speed=speed_bak;
```

恢复速度为初始值。

```
    longdelay(tab[i*2+1]);
```

延迟，按照索引查表。

```
    }
  }
```

整个模式运行完毕。

```
    heat_var=0;
    stop();
```

关闭加热，停止电动机运转。

```
    }
```

2.8.6 定位程序

定位程序如下：

```
void position()
{
    unsigned int cnt=1;
```

定位程序的作用是模拟计数变量，实现长时间的计数。

```
    speed=SLOW;//run moto
```

速度下降为最低挡。

```
    TH0=0;
```

清零定时器 0。

```
while(SN2==0) {
```

如果槽形光耦没有被遮光板挡住。

```
if(TH0>120) {cnt+=1;TH0=0;}
```

每当定时器 0 的 TH0 达到 120 的时候，清零自己，并且 cnt 增加，用此方法实现 cnt 的增加。

```
if(cnt>150) break;
```

如果 cnt 超过 150，则退出。这用于万一槽形光耦的遮光板出现故障（经常发生折断），可自动停止。

```
};  
delay(70);
```

当遮光板进入的时候，延迟，防止干扰。

```
TH0=0;
```

清零 TH0。

```
cnt=1;
```

重置 cnt 为 1。

```
while(SN2==1) {
```

只要遮光板在槽形光耦中就运行以下程序。

```
if(TH0>120) {cnt+=1;TH0=0;}
```

每当定时器 0 的 TH0 达到 120 的时候，清零自己，并且 cnt 增加，用此方法实现 cnt 的增加。

```
if(cnt>150) break;
```

如果 cnt 超过 150，则退出。这用于万一槽形光耦的遮光板出现故障（螺钉滑落，遮光板一直在槽形光耦中），可自动停止。

```
};
```

2.8.7 主控子程序

本程序首先是键盘处理，读取按了哪个键，然后对显示面板上的 LED 进行点灯，接着按照加热选择设置电热丝的 PWM 数值，最后按照电动机运转选择模式设置电动机运转的 PWM 数据。


```
int mysub(){  
    keyscan();
```

扫描键盘。

```
    if(run_tag==0)  
    {
```

如果目前是停止标志，

```
        hot=NO_HOT;
```

则加热清零。

```
        speed=NO_SPEED ;
```

速度清零。

```
    }  
    switch(heat_var)
```

根据加热变量处理。

```
    {  
        case 0:
```

如果不用加热。

```
            hot=NO_HOT;
```

则电热丝的 PWM 参数 hot 设置为 0。

```
            HOT_Hi=0;
```

高温对应的发光指示灯亮。

```
            HOT_Lo=0;
```

低温对应的发光指示灯亮。

```
        if(pgm_style==0) run_tag=0;
```

如果运行的电动机工作模式为 0，则停止标志清零，表示停止。

```
        break;
```

```
        case 1:  
            hot=MID_HOT;
```

电热丝的 PWM 参数 hot 设置为 MID_HOT。

```
            HOT_Hi=1;
```

高温对应的发光指示灯灭。

```
HOT_Lo=0;
```

低温对应的发光指示灯亮。

```
run_tag=1;
```

运行标志置位。

```
break;

case 2:
    hot=HIGH_HOT;
```

电热丝的 PWM 参数 hot 设置为 HIGH_HOT。

```
HOT_Hi=0;
```

高温对应的发光指示灯亮。

```
HOT_Lo=1;
```

低温对应的发光指示灯灭。

```
run_tag=1;
```

运行标志置位。

```
break;
}

switch(pgm_style)
```

根据运行电动机模式选择。

```
{
case 0:
```

电动机停止模式。

```
PGM_I=0;
```

间断模式指示灯灭。

```
PGM_II=0;
```

持续模式指示灯灭。

```
FAST_SLOW_I=0;
```

低速指示灯灭。

```
FAST_SLOW_II=0;
```

中速指示灯灭。

```
FAST_SLOW_III=0;
```

高速指示灯灭。

```
if(heat_var==0) run_tag=0;
```

如果加热变量为 0，则停止标志清零。这种情况是电动机可以不转，仅电热丝加热，但当电热丝停止加热时，运行进入停止状态。

```
break;
```

```
case 1:
```

间断模式。

```
PGM_I=1;
```

间断模式指示灯亮。

```
PGM_II=0;
```

持续模式指示灯灭。

```
run_tag=1;
```

运行标志置位。

```
break;
```

```
case 2:
```

电动机持续运转模式。

```
PGM_I=0;
```

间断模式指示灯灭。

```
PGM_II=1;
```

持续模式指示灯亮。

```
run_tag=1;
```

运行标志置位。

```
break;
```

```
}
```

```
if(pgm_style!=0) {
```

如果电动机运转模式不是停止，

```
switch(fast_slow_var)
{
```

则根据转速变量选择。

```
case 0:
```

低速。

```
FAST_SLOW_I=1;
```

低速指示灯亮。

```
FAST_SLOW_II=0;
```

中速指示灯灭。

```
FAST_SLOW_III=0;
```

高速度指示灯灭。

```
speed=SLOW;
```

电动机速度 PWM 设置数值为 SLOW。

```
break;
```

```
case 1:
```

中速情况。

```
FAST_SLOW_I=0;
```

低速指示灯灭。

```
FAST_SLOW_II=1;
```

中速指示灯亮。

```
FAST_SLOW_III=0;
```

高速指示灯灭。

```
speed=MID;
```

速度 PWM 设置数值为 MID。

```
break;
```

```
case 2:
```

高速情况。

```
FAST_SLOW_I=0;
```

低速指示灯灭。

```
FAST_SLOW_II=0;
```

中速指示灯灭。

```
FAST_SLOW_III=1;
```

高速度指示灯亮。

```
speed=FAST;
```

速度 PWM 设置数值为 FAST。

```
break;} } }
```

2.8.8 主控程序

主函数 main 如下：

```
main()
{
    unsigned long int i;
```

定义长整型 i。

```
P1M1=0x0;
P1M0=0xff;
```

初始化 P1 口配置寄存器，对输入/输出进行设置。

```
P3M1=0x7b;
P3M0=0x80;
```

初始化 P3 口配置寄存器。

```
P1_7=0;
P1_0=0;
P1_1=0;
P1_3=1;
P1_2=1;
P1_4=1;
```

设置 I/O 的初始状态。

```
BELL=0;
for(i=0;i<60000;i++);
```

```
BELL=1;
```

蜂鸣器响一下。

```
speed=0;
```

电动机速度为 0。

```
TMOD=0x21;
```

初始化定时器配置。

```
TH1=0xEA;    //240  
TL1=0xD0;
```

定时器 1 初始化。

```
TR1=1;  
TR0=1;  
TI=1;  
ET1=1;  
IT0=1;
```

定时器 0 中断允许。

```
EX0=1;
```

外中断允许。

```
EA=1;
```

开中断。

```
i=0;  
while(1)  
{  
    mysub();
```

调主控制函数。

```
if(need_position==1)
```

如果用户设置为停止，

```
{  
    stop();
```

则电动机和加热停止。

```
need_position=0;
```

定位标志清零。

```
}
```

```
switch(pgm_style)
```

根据电动机模式选择。

```
{
```

```
case 0:
```

不运转情况。

```
break;
```

```
case 1:
```

间断模式。

```
running_mode_1();
```

调间断模式子程序。

```
break;
```

```
case 2:
```

持续模式。

```
longdelay(100);//delay 10s
```

延迟。

```
i++;
```

软件计数器 i 增加。

```
if(i>97)
```

当 i 达到 97 的时候，

```
{i=0;
```

清零 i。

```
heat_var=0;
```

清零加热变量。

```
stop();
```

电动机停止，面板显示停止 LED。

```
}
```

```
break;  
}}}
```

至此，足疗机的设计介绍完毕。

从第 3 章开始，将在本章的基础上更加深入一步，介绍乒乓自动发球机的设计。乒乓自动发球机也叫乒乓陪练机，可以在一个人的情况下和机器打乒乓球。

第 3 章

更进一步——乒乓发球机产品设计

第 2 章讲述了最基本的单片机产品开发，本章将进一步深入，设计一个相对复杂但还算比较简单的产品——桌面式乒乓自动发球机。具体包括两大部分：红外遥控器和控制主板。红外遥控器涉及的设计有液晶显示接口、红外数据发射、键盘、休眠、低功耗模式；控制主板涉及舵机控制、PWM 电动机控制、红外数据接收。由于采用的单片机资源有限，因此如何合理安排软硬件资源成了本设计的关键内容。

通过本章的学习将掌握以下内容：

- 电动机正反转控制线路——H 桥；
- 舵机控制方法；
- 反射式红外检测技术；
- 单片机语音播放的实现；
- 电动机过流检测方法；
- 调制式红外发射；
- 点阵液晶显示接口。

3.1 需求分析

首先还是分析买来的样机的功能，然后在此基础上设计自己的产品。图 3-1 是发球机的使用环境，发球机安装在乒乓球台上，在后方有收集网，机器发过来的球被人打过去以后落在收集网内，最终落到发球机的供球盒，在供球电动机的作用下依次被逐渐提升后通过发球电动机发射出来。球台分为 N 个发球落点，可以通过对发球机进行设置实现。



图 3-1 自动发球机使用环境

该自动发球机一共有 7 个落点，用户可通过红外遥控器设置好参数，然后通过红外遥控来命令发球机启动发球。

从图 3-2 可以看出，发球机由最底下的供球电动机将球输送到最上方的发球头，发球摆头装在舵机上，舵机则安装在机身梁上。因此前方的摆头是在舵机的控制下转动的。舵机通过左右摆动控制当前发球的落点，发球摆头内部有上下两个电动机，电动机轴上装胶轮，通过和球发生摩擦把球发出去。可以分别控制上下电动机的速度，如果上下的速度一样，球平飞出去；当下面的电动机快，上面的电动机慢时，球向下压着发出；当下面的电动机慢，上面的电动机快时，球向偏上方发出。上下速度差越大，发出的球角和水平夹角也越大。发球的电动机速度越高，球发出去的速度越快，力量也越大。

要设计同类产品，最重要的是分析市场，分析竞争的优势在哪里。如果完全做和样品一模一样的产品，首先是有知识产权侵权的风险，然后是销售优势并不具备。其他公司已经在这款产品上有了销售渠道和经验，你新做出来的如何能确保销售比他人顺利？最后肯定是打价格战，很可能生产的产品没有利润。因此完全模仿不是正常的发展方式。有不少公司为了降低研发投入采用抄袭的方式仿制产品，这除了有法律风险以外，也谈不上持续发展。

另外，不同的产品销售渠道和模式不同，比如前面提到的足疗机和本章的乒乓自动发球机相比，销售方式完全不同。足疗机走的是普通保健电子产品的销售模式，客户面向的是普通百姓，要靠量大，因此定价不能高。乒乓自动发球机面向的是行业用户，一般销售给俱乐部、体育馆、国外的家庭健身用。因此乒乓发球机的量不会有足疗机那么大，所以定价也要适当提高，故在设计中要显示一定的档次。

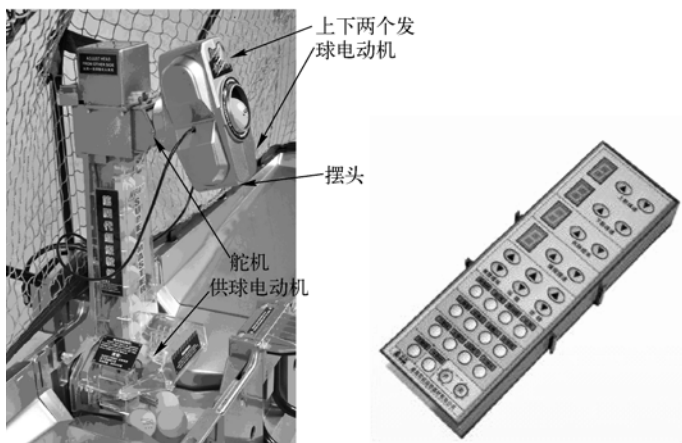


图 3-2 样品机产品图片——主机和遥控器

分析目前的样品机并结合对使用客户的调查,发现目前其他公司生产的发球机的遥控器使用不方便,遥控器是有线遥控方式,有一根很长的线,打乒乓球时用起来不方便。其次就是现有的遥控器采用数码管方式,要设置复杂点的参数很不方便。因此综合各方面的意见,修改现有的有线遥控方式为红外遥控方式,修改 LED 八段数码管方式为液晶显示方式。主机整体功能不变,适当增加辅助功能,如语音提示。

下面对发球机的两个主要部分——遥控器和发球主机的详细需求进行介绍。

3.1.1 遥控器需求分析

遥控器(单向传输,遥控距离不小于 10m):

- (1) 液晶显示,当按住按键不松手时,数字应该连续变化。
- (2) 当光标停在语种选择的“简体”上时,按“+”或“-”键可以变成“繁体”、“English”,同时将屏幕上的所有字符变成与之对应的语种。
- (3) 当光标停在计数的“9999”上时,按“+”或“-”键可以改变此数值(范围为 1~9999,倒计时),每按一下变化 100;当数值小于 100 后,每按一下变化 5;当数值小于 10 后,每按一下变化 1。
- (4) 当光标停在计时的“99”上时,数值变化范围为 1~99,倒计时。
- (5) 当光标停在上轮速度的“3”上时,数值变化范围为 0~9。
- (6) 当光标停在无规律发球的“关”上时,变化范围为开-关。当是“开”时,落点位置、个数、出球频率三者要锁住不可调。
- (7) 当光标停在下轮速度的“3”上时,数值变化范围为 0~9。
- (8) 上轮速度和下轮速度不能同时为“0”。
- (9) 当光标停在落点位置的“7”上时,数值变化范围为 1~11。
- (10) 当光标停在个数的“2”上时,数值变化范围为 0~99。
- (11) 当光标停在出球频率的“5”上时,数值变化范围为 1~9。
- (12) 当按下遥控器上的“暂停”键时,自动保存液晶屏幕上的所有信息,关闭无规律发球,同时发球机进入暂停状态。

(13) 当按下遥控器上的“开始”键时，主板按照液晶屏幕上的命令运行。

(14) 当发球机处于运行状态时，从遥控器上“自动卸球”键发过来的命令主板不予执行；只有当发球机处于暂停状态时主板才执行（供球电动机顺时针以5挡的速度转5圈，即光电开关感应15下）。

3.1.2 主板需求分析

1) 当主板接通电源时，发球机处于暂停状态（计时也暂停）。

2) 开始状态：也叫运行状态，表现如下。

(1) 发球机按照遥控器上的命令运转（供球电动机要晚2s启动）。

(2) 开始倒计时，当主板上的计数用完时（为0时），此时发球机自动进入暂停状态。

(3) 当无规律发球是“开”时，落点位置、个数、出球频率三者要随机变化（但个数不能为0）。

(4) 发球机处于开始状态时，从遥控器发过来的“无规律发球”、“落点位置”、“个数”、“出球频率”四者命令主板不予执行；只有当发球机处于暂停状态，按了“开始”键后主板才执行。

(5) 落点位置、个数、出球频率三者关系举例：落点位置3、个数2、出球频率5，表示以5挡的出球频率在落点3的位置上发2个球。执行完这一行命令再执行下一行命令，如此反复循环执行这11行命令（当个数为0时直接跳到下一行执行）。

3) 主板上要有卡球倒转（用检测电阻方式）、语音报数报字装置。

详细的电动机输出转速要求如表3-1所示。

表 3-1 电动机输出转速

	1 挡	2 挡	3 挡	4 挡	5 挡	6 挡	7 挡	8 挡	9 挡
上轮	3 400	4 300	5 300	6 000	6 500	7 400	8 100	8 600	9 200
下轮	2 500	2 800	3 400	3 900	4 200	4 700	5 300	5 900	6 400
供球	13	17	21	25	29	33	37	41	45

根据市场需求分析结果，决定采用如图3-3所示的遥控器按键布局。

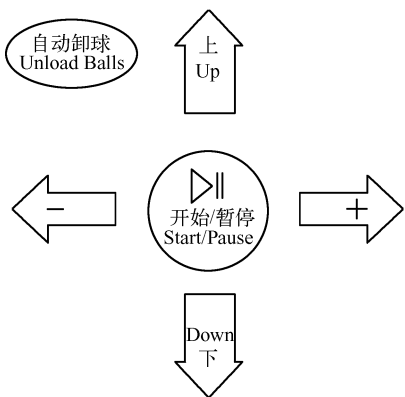


图 3-3 遥控器按键布局

中间的按钮是启动/停止按钮，左边的是减少按钮，右边的是增加按钮，上面的是光标上移按钮，下面的是光标下移按钮，左上方是自动卸球按钮。

遥控器面板上的液晶显示器的显示要求如图 3-4 所示，左边是简体，中间是繁体，右边是英语。程序中应设计三种显示方式可自由切换。

上轮速度9			上輪速度9			Topspin speed 9
下轮速度9			下輪速度9			Backspin speed 9
落点3	个数2	频率5	落點3	個數2	頻率5	Point3 No.2 Freq5
落点3	个数99	频率9	落點3	個數99	頻率9	Point11 No.99 Freq9
落点3	个数99	频率9	落點3	個數99	頻率9	Point11 No.99 Freq9
落点3	个数99	频率9	落點3	個數99	頻率9	Point11 No.99 Freq9
落点3	个数99	频率9	落點3	個數99	頻率9	Point11 No.99 Freq9
落点3	个数99	频率9	落點3	個數99	頻率9	Point11 No.99 Freq9
落点3	个数99	频率9	落點3	個數99	頻率9	Point11 No.99 Freq9
落点3	个数99	频率9	落點3	個數99	頻率9	Point11 No.99 Freq9
落点3	个数99	频率9	落點3	個數99	頻率9	Point11 No.99 Freq9
落点3	个数99	频率9	落點3	個數99	頻率9	Point11 No.99 Freq9
落点3	个数99	频率9	落點3	個數99	頻率9	Point11 No.99 Freq9
落点3	个数99	频率9	落點3	個數99	頻率9	Point11 No.99 Freq9
无规律发球：关			無規律發球：關			Irregular: OFF
计数：9999			計數：9999			Counter: 9999
计时：99分钟			計時：99分鐘			Timer: 99min
语种选择：简体			語種選擇：繁體			Language: English

图 3-4 遥控器面板的显示要求

现在按照上面提到的需求，分析整机的硬件设计方案。由于需要对落点进行准确的设定，因此决定采用舵机完成此功能。由图 3-2 可知发球机还需要供球电动机，供球电动机要求力量较大，因此选择具有减速功能的电动机做供球电动机。另外，摆头上的发球电动机需要上下两个，这两个电动机只是通过轴上的胶轮把球高速带出去，不需要很大的力，但要求速度快，所以不需要减速功能。因此用到的电动机一共是：1 个舵机、1 个减速电动机、2 个发球电动机。其中减速电动机是在普通 12V 电动机的基础上通过减速箱齿轮减速的。图 3-5 和图 3-6 是发球电动机的外观和参数。

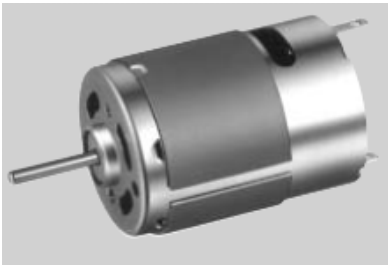


图 3-5 发球电动机的外观

型号	电压		空载		最大效率				堵转		
	范围	正常	速度	电流	速度	电流	力矩		输出	力矩	
			r/min	A	r/min	A	mNm	gcm	W	mNm	gcm
RS-380PH-3270 (*1)	4.5~15.0	12V CONSTANT	16400	0.37	14110	2.28	13.0	133	19.2	93.2	950

图 3-6 发球电动机的参数

3.2 硬件功能设计和实现

3.2.1 落点的实现

由于有不同的落点，为了让发球头的位置在不同发球时刻处于规定的位置，可选用舵机来实现。图 3-7 是舵机和发球摆头的形状。



图 3-7 舵机和发球摆头

从图 3-7 中可以看出，黑色发球摆头安装在舵机上，舵机在白色外壳的内部。随着舵机的运动，发球头可以按一定的角度左右摆动。为了理解舵机的工作，先介绍舵机的结构和内部原理，如图 3-8 和图 3-9 所示。

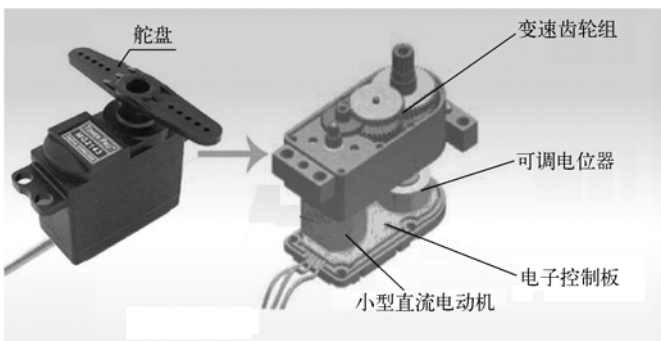


图 3-8 舵机结构

舵机是一种位置伺服的驱动器，主要由外壳、电路板、无核心电动机、齿轮与位置检测器构成。其工作原理是由接收机或者单片机发出信号给舵机，内部有一个基准电路，产生周期为 20ms、宽度为 1.5ms 的基准信号，将获得的直流偏置电压与电位器的电压比较，获得电压差并输出。经由电路板上的 IC 判断转动方向，再驱动无核心电动机开始转动，通过减速齿轮将动力传至摆臂，同时由位置检测器送回信号，判断是否已经到达定位。舵机适用于那些需要角度不断变化并可以保持的控制系统。当电动机转速一定时，通过级联减速齿轮

带动电位器旋转，使得电压差为零，电动机停止转动。一般舵机旋转的角度范围是 $0^{\circ} \sim 180^{\circ}$ 。

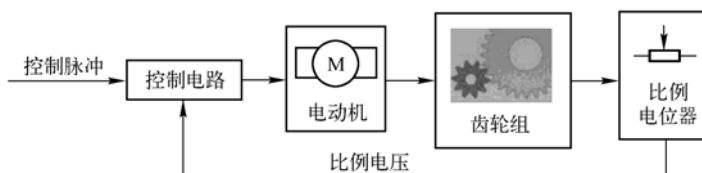


图 3-9 舵机原理

舵机有很多规格，但所有的舵机都外接有三根线，分别用棕、红、橙三种颜色进行区分。由于舵机品牌不同，颜色也会有所差异，棕色为接地线，红色为电源正极线，橙色为信号线。

舵机的参数如下。

- 转速：由舵机无负载的情况下转过 60° 角所需时间来衡量，常见舵机的速度一般为 $0.11\text{s}/60^{\circ} \sim 0.21\text{s}/60^{\circ}$ 。
- 扭矩：单位是 $\text{kg}\cdot\text{cm}$ ，这是一个扭矩单位，可以理解为在舵盘上距舵机轴中心水平距离 1cm 处，舵机能够带动的物体质量。
- 电压：小型舵机的工作电压一般为 4.8V 或 6V 。
- 质量：以 g 为单位，有微型 9g 舵机，中型 45g 、 100g 舵机等。

舵机的转动角度是通过调节 PWM（脉冲宽度调制）信号的占空比来实现的。参考图 3-10，标准 PWM 信号的周期固定为 20ms （ 50Hz ），理论上脉宽分布应在 $1\sim 2\text{ms}$ 之间，但是，事实上脉宽可在 $0.5\sim 2.5\text{ms}$ 之间，脉宽和舵机的转角 $0^{\circ} \sim 180^{\circ}$ 相对应。有一点值得注意，由于舵机的品牌不同，对于同一信号，不同舵机旋转的角度也会有所不同。

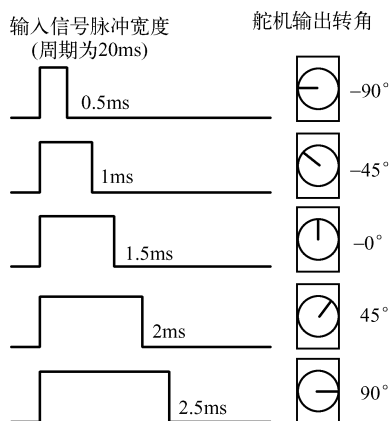


图 3-10 舵机转角和输入脉冲的关系

注意：这只是一种参考数值，具体的参数请参见舵机的技术参数。改变高电平的脉冲宽度就改变了输出角度。

舵机只有 3 根线：电压，地，脉宽控制信号线。与单片机的接口只需要一条线，用单片机的定时器产生 20ms 的脉冲频率控制舵机，通过改变脉冲的占空比来控制输出角度。舵机转动时需要消耗比较大的电流，所以舵机的电源最好单独提供，不要和单片机使用同一路电源。否则极易发生抖动，影响产品性能。为此，可以在舵机供电的电源线那里专门用一个功率稍微大点的电源稳压芯片。

根据发球头上的总质量，综合考虑落点切换速度等因素，选用以下的舵机，见图 3-11。舵机可根据设计参数让工厂定做。



图 3-11 自动发球机选用舵机

- 型号：DYS0210
- 重量：40g
- 尺寸：40.8×20.1×38.0 (mm³)
- 速度：0.16s/60°
- 力矩：6.0kg/cm
- 工作电压：4.80~6V

3.2.2 发球个数的实现

由于要对每个落点上所发球的个数进行统计，因此需要一个传感器来计算发球个数。从图 3-12 中可以看到，供球转盘有 3 个叉，每个叉负责运送一个球。

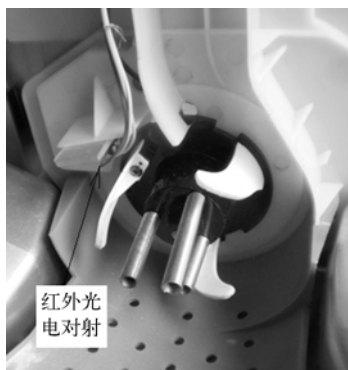


图 3-12 红外计数

这 3 个叉随着转盘循环旋转不断把球输送上去。叉的颜色是白色的，因此可以在左上方安装一个红外的计数传感器，每当白色的物体过来，传感器就给出一个信号。这里选用 TCRT5000 反射式光电传感器，如图 3-13 和图 3-14 所示。这种红外光电传感器可用于检测物体的有无、进行定位计数、测速等。其产品特点是：耐高压、高阻抗、电器隔离性能好、抗干扰能力强。



图 3-13 TCRT5000 外观

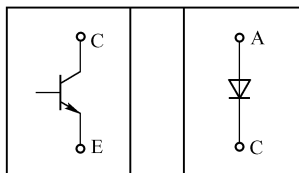


图 3-14 TCRT5000 内部结构

从图 3-15 可以看到，反射光电开关 TCRT5000 分为发射和接收两个部分。发射部分通电流 I_F ，当有反射发生时，接收管电流为 I_C ，当没有反射发生时，接收电流非常小，接近零。因此通过电流可以知道是否有反射物体。

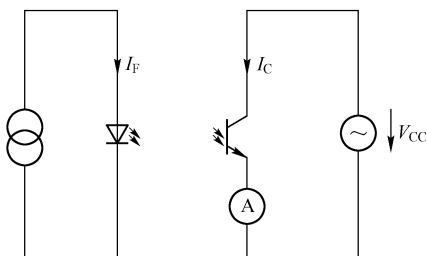


图 3-15 反射光电开关工作原理

本设计中具体的实现如图 3-16 所示，P4 的 3 脚接 CTRT5000 发射管的阳极，1 脚接发射管的阴极。P4 的 1 脚接 TCRT5000 接收管的 e，P4 的 2 脚接 TCRT5000 接收管的 c。当有反射发生时，P4 的 2 脚电平发生改变，单片机 U4 的外中断引脚接收到信号。

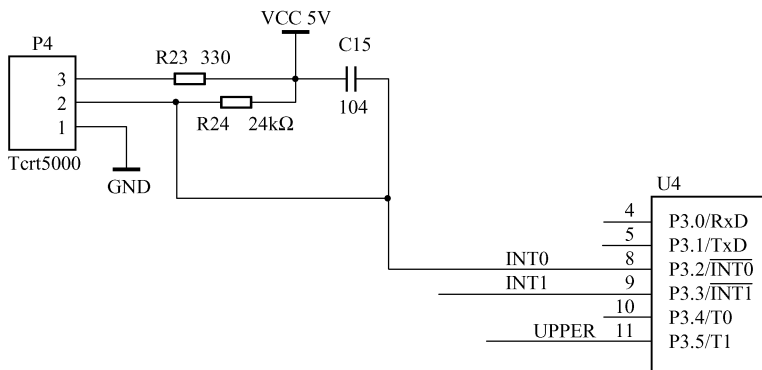


图 3-16 具体的实现方式

3.2.3 红外接收的实现

为了简化红外接收的实现，红外接收选用一体化的集成接收电路 VS1838B，这是一款性价比很高的产品，见图 3-17 和图 3-18。如果不用集成模块，则要另外设计滤波、整形、抗干扰等很多外围电路，不但增加了成本而且容易不稳定，因此不推荐。

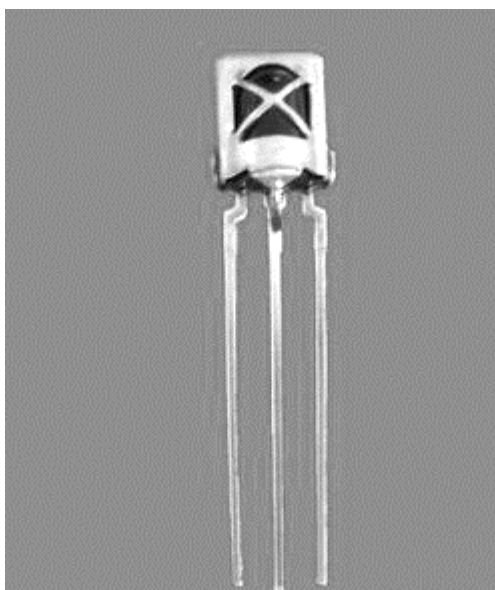


图 3-17 VS1838B 红外接收

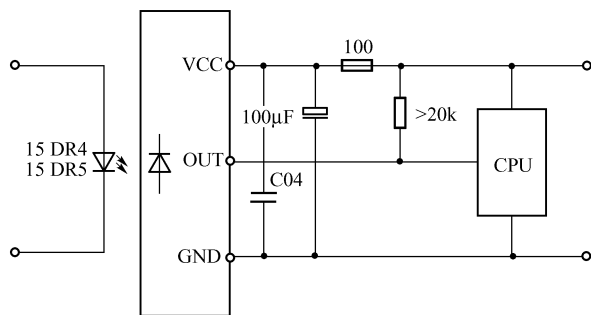


图 3-18 VS1838B 典型应用图

在本设计中，VS1838B 的正端接到 P1 的 3 脚，负端接到 P1 的 2 脚，输出接到 P1 的 1 脚，通过 R13 上拉电阻，送到单片机的中断 1，如图 3-19 所示。

VS1838B 有一个遥控的角度和距离，图 3-20 中有详细描述，因此遥控器要在此范围内才正常。如果不注意这一点，设计出来的产品就会有接收不正常的现象。

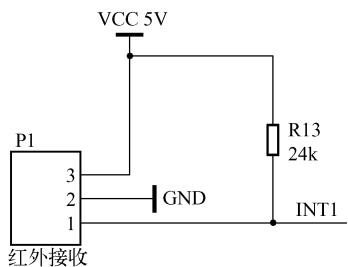


图 3-19 VS1838B 的连接

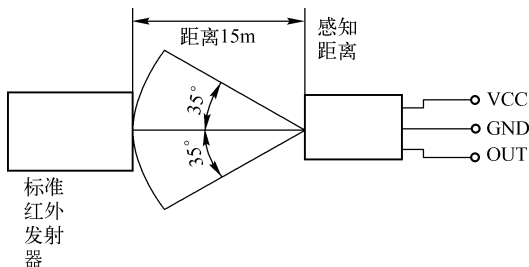


图 3-20 VS1838B 遥控角度和距离

遥控器的发射波形要按照 VS1838B 的要求来做，否则会出现接收不稳定和距离短的问题，见图 3-21。

如果发射的占空比或者频率和图 3-21 所示的偏差较大，就会发现本来可以发射 10m 的，现在只能发射 2m。用示波器观察接收波形，发现和发射波形有偏离。因此要严格按照手册的要求来设计。

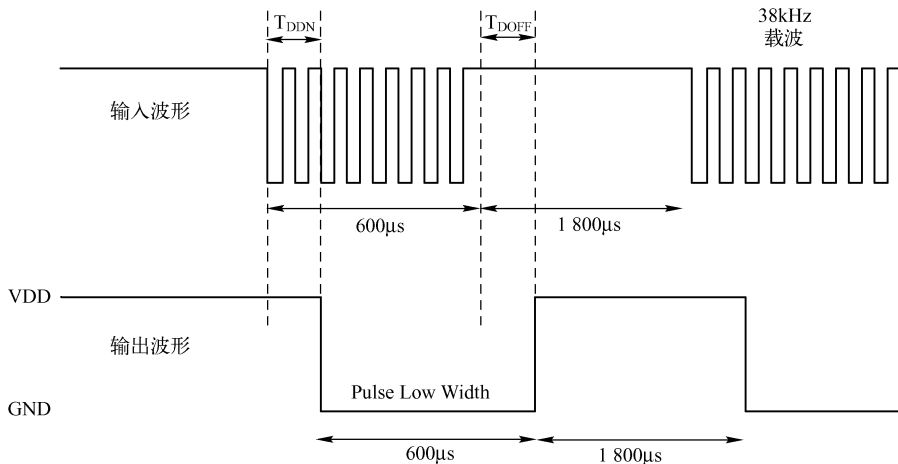


图 3-21 VS1838B 测试波形

3.2.4 供球电动机正反转和调速的实现

由图 3-2 可以看到，乒乓球在发球机下方通过供球电动机输送上去，需求中也提到，一旦卡球，电动机自动倒转，释放所卡之球。为此需要设计支持电动机正反转的电路。可以选择现成的电动机正反转集成电路，也可以用分立元件实现，各有优缺点。集成电路实现方便，但不容易找到 12V 电压的正反转控制芯片；另外，集成电路生产调试方便，但是成本偏高。而用分立元件做的肯定成本低，但调试麻烦，容易出错。不过对于设计成熟并大批量生产的产品来说，只要是机器贴片，一般一批产品出来并不会出什么问题，唯一可能出问题的是虚焊。所谓电动机正反转电路其实就是一个 H 桥。下面首先简单介绍一下 H 桥的基本原理，如图 3-22 所示。

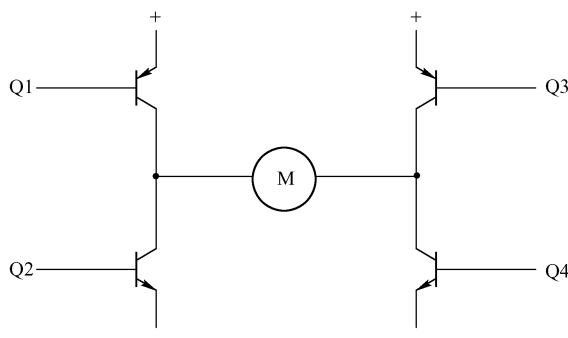


图 3-22 H 桥的基本原理

H 桥驱动电路的名称缘于它的形状酷似字母 H。4 个三极管组成 H 的 4 条垂直腿，而电动机就是 H 中的横杠。H 桥式电动机驱动电路包括 4 个三极管和一个电动机。要使电动机运转，必须导通对角线上的一对三极管。根据不同三极管对的导通情况，电流可能会从左至右或从右至左流过电动机，从而控制电动机的转向。

如图 3-23 所示，当 Q1 管和 Q4 管导通时，电流从电源正极经 Q1 从左至右穿过电动机，然后再经 Q4 回到电源负极。按图中电流箭头所示，该流向的电流将驱动电动机沿顺时针方向转动。

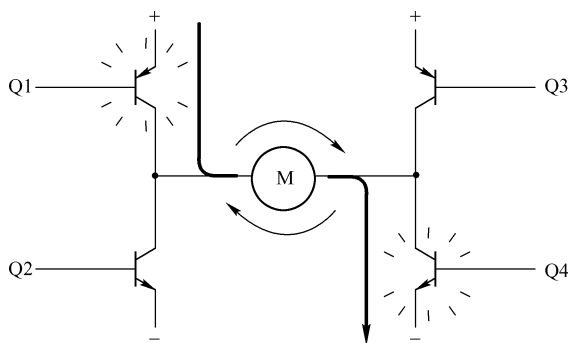


图 3-23 通过 H 桥电动机正转

在图 3-24 中，当三极管 Q2 和 Q3 导通时，电流将从右至左流过电动机，从而驱动电动机沿逆时针方向转动。

驱动电动机时，保证 H 桥上两个同侧的三极管不会同时导通非常重要。如果三极管 Q1 和 Q2 同时导通，则电流会从正极穿过两个三极管直接回到负极。此时，电路中除了三极管外没有其他任何负载，因此电路上的电流可能达到最大值（该电流仅受电源内电阻的限制），如果电源电压较高则会烧坏三极管。基于上述原因，在实际驱动电路中通常要用硬件电路方便地控制三极管的开关。

图 3-25 是从网上找到的低压小电流 H 桥电路，只适合玩具小电动机，经常在玩具车中看到此类电路。有人会问，为什么玩具中不采用集成电路来实现呢？原因很简单，玩具要求生产价格越低越好。

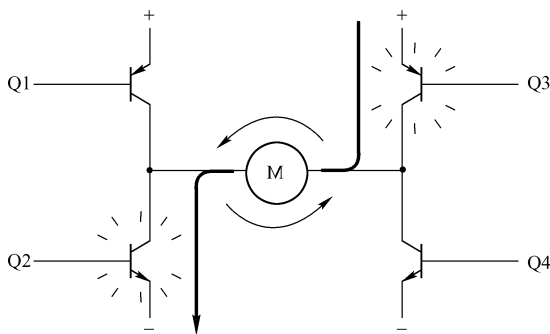


图 3-24 通过 H 桥电动机反转

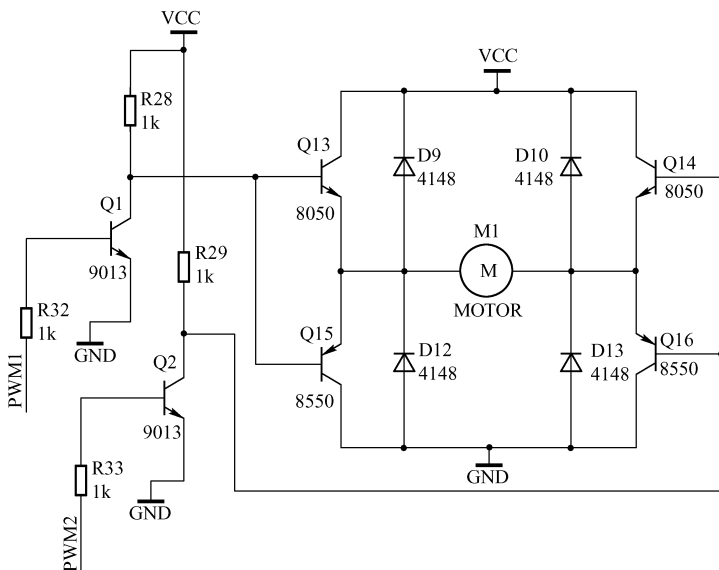


图 3-25 低压小电流 H 桥电路

Q13、Q14、Q15、Q16 组成 H 桥主开关。D9、D10、D12、D13 是续流管，保证在开关管关闭器件电动机后电流能继续。如果没有续流管，电动机转子的电感会产生高压，可能击穿三极管。根据 $U=L \cdot di/dt$ ，如果电流发生突变， di/dt 会很大，将超过三极管最大承受电压。Q13 和 Q15 组成上下一对开关，Q13 是 NPN 管，Q15 是 PNP 管。当 Q13 的基极为高时，Q13 导通，Q15 关闭；反之 Q13 关闭，Q15 导通。因此始终只有一个管导通，不会发生直通的问题。Q1 和 Q2 是驱动管，将来自单片机的信号放大，保证有足够的电流驱动 H 桥。到目前为止大家看出图 3-25 中的问题没有？其实图 3-25 是不对的，问题出在 H 桥的 4 个开关管上。分析 Q13 可以看到，Q13 的 e 端没有固定的电压，因此 Q13 的 b 和 e 之间并没有确定的压差，导致的问题是 Q13 无法进入开关状态。为此，必须将 Q13 和 Q15 对换，将 Q14 和 Q16 对换。

图 3-26 是修正后的 H 桥。再仔细看看，还有问题吗？还有问题，Q13 和 Q14 的驱动已经正常，但 Q15 和 Q16 的基极驱动有问题。如果 Q1 导通，电流将从 VCC 到 Q15、再到 Q1，中间没有限流电阻，因此基极电流会损坏三极管 Q1 和 Q15。

图 3-27 是修改正确的 H 桥。这里强调一点，必须理解电路，切忌在不理解的情况下从网上直接下载使用，因为很多都是错误的！

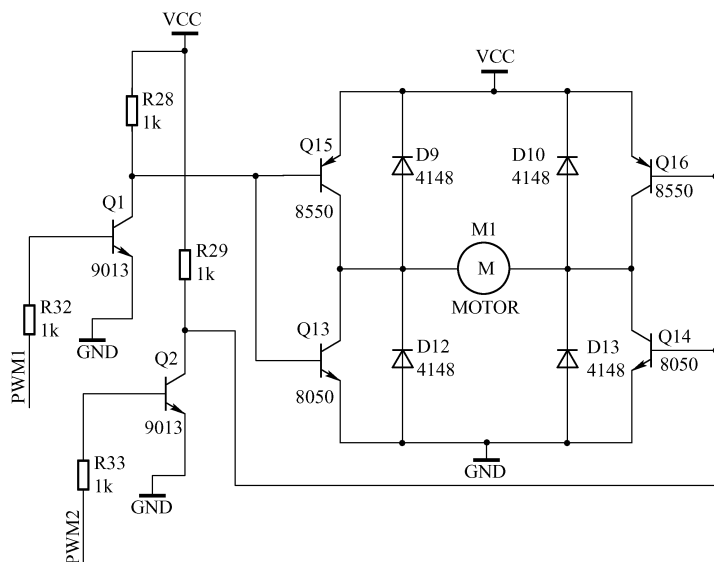


图 3-26 修正后的 H 桥

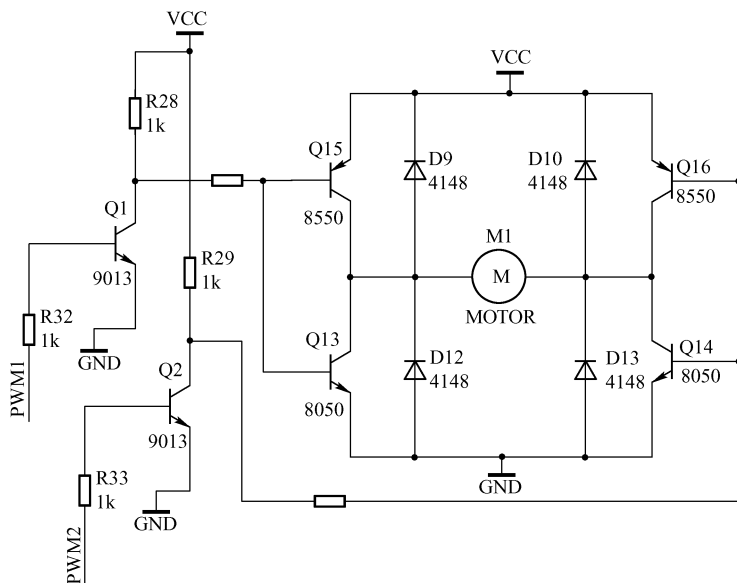


图 3-27 正确的 H 桥

H 桥开关管处于饱和和导通状态很关键，所以要注意三极管基极限流电阻的阻值，如果选择过大，会导致三极管工作在放大状态，由于电动机的电阻非常小，因此很快会将三极管烧毁。另外，不同的电动机对应不同的基极电流，总之最关键的一点就是三极管一定要在饱和状态下工作。如果工作在放大状态，如 ce 压降为 $5V$ ，电流为 $300mA$ ，则三极管自身消耗功率为 $1.5W$ ，很快就会报废；如果工作在饱和状态，假设 ce 压降为 $0.3V$ ，电流为 $300mA$ ，则三极管自身消耗为 $0.09W$ ，问题不大。还需要注意的一个问题是开关频率问题。三极管从导通到截止，中间一定要经过放大区，只是时间非常短，不会造成不良影响。但是，当频率很高时，这

个放大区的过渡次数就很频繁，在大电流的时候三极管的发热也越发明显，为此频率不能过高。当然，频率过低也有问题，最可能发生的问题是电动机因为机械共振严重抖动；另外，频率在 $1\sim 3\text{kHz}$ 时人耳会听到尖叫声。当然，在只需要控制正反转而不需要进行 PWM 调速的场合不存在这个问题。三极管作为 H 桥主开关器件的问题是饱和压降偏大，为了降低器件本身的发热，最好选用 MOS 管作为主开关器件。图 3-28 是本项目 H 桥的实现。

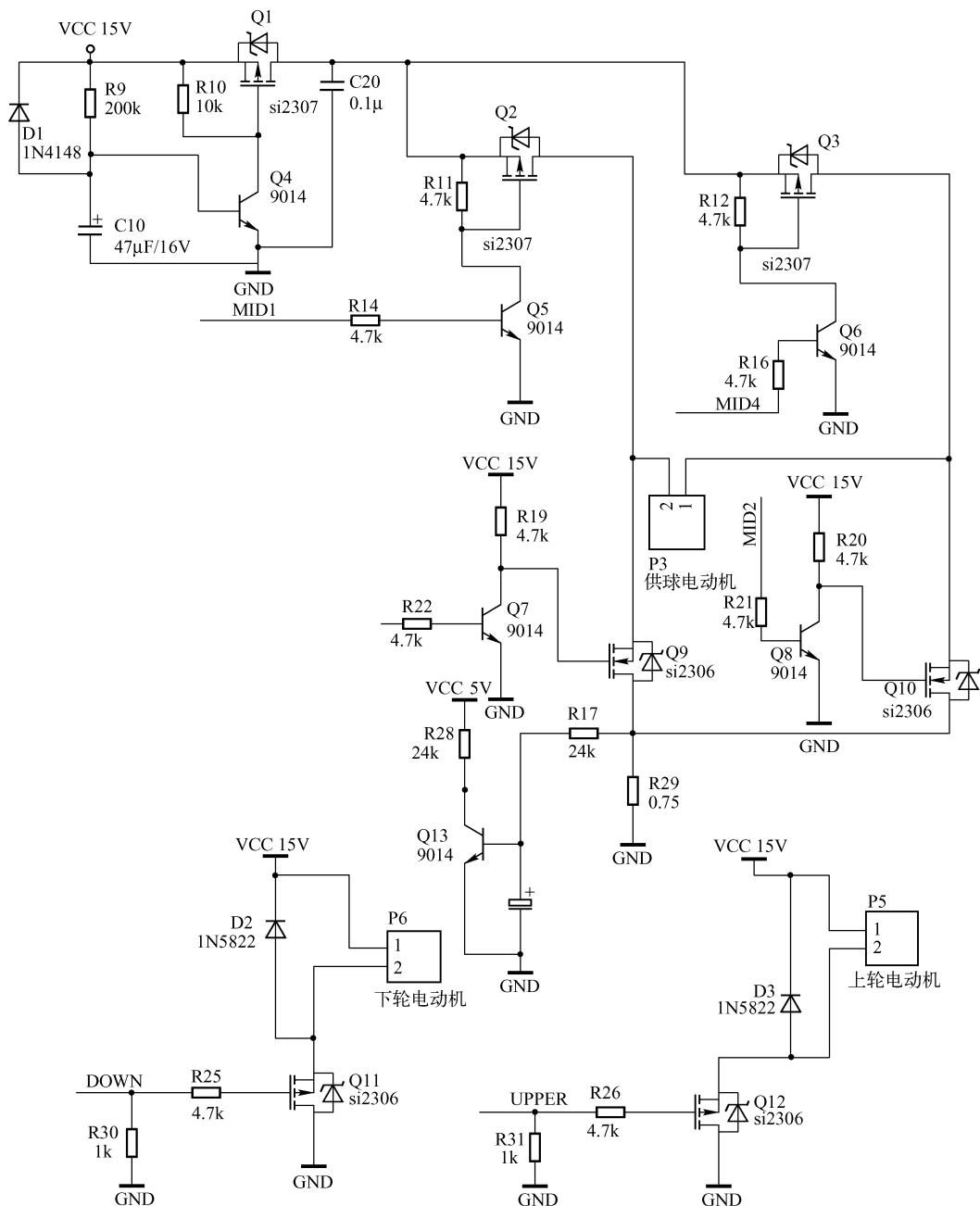


图 3-28 H 桥的实现

本设计采用分立元件实现 H 桥。Q2、Q3、Q9、Q10 构成 H 桥主开关器件，Q5、Q6、Q7、Q8 为 MOS 的驱动管，用以隔离功率 MOS 和单片机。分析 Q2 的驱动，R11 是上拉电阻，当 Q4 截止时，提供 MOS 管门极高电平，保证其处于截止状态，避免进入放大和不确定状态而损坏 MOS 管。当 MID1 上是高电平时，Q5 导通，Q2 的门极为低电平，Q2 导通。Q10 是 N 型 MOS 管，当 MID2 为低电平时，Q8 截止，Q10 的门极为高电平，Q10 导通。电流从 Q2 通过电动机到 Q10，再到 R29、到地。R29 用于检测电流过流。由于需要给定电动机的速度，因此可以让 Q2 导通，而 Q10 用 PWM 波控制，这样可调整输出电压。

单片机的 4 个引脚控制这 4 个 MOS 管的状态，因此编写程序时一定要仔细，千万不可让上下直通，直通会烧毁 MOS 管。但是，在系统上电的瞬间，单片机还未工作，从复位到进入正常工作期间，单片机的引脚状态不稳定，会导致直通。因此在图 3-28 中加了 Q1，让电压过若干时间再提供到 H 桥臂上。由图中可以看到，上电开始时 C10 未充电，Q4 的基极为低电平，Q4 截止，Q1 不导通。随着时间的推移，R9 向 C10 充电，Q4 的基极电流慢慢升高，到一定程度时 Q4 导通，Q4 的集电极为低电平，Q1 导通，给 H 桥提供电源。当突然掉电时，C10 通过 D1 迅速放电，Q1 立刻关闭，防止掉电过程中因为 MOS 门极电压不稳定而进入放大状态，瞬间损坏 MOS 管。从图 3-28 中可以看到，采用 MOS 管后，由于 MOS 本身存在反向二极管，因此可以不外加续流管。但是如果电流很大则还是要外加，否则会发热烧毁。

图 3-28 中采用的 MOS 管型号是 si2306 和 si2307，按照手册的说明，上图不存在问题，因此买原装进口的就可以正常运行。而在市场上常常会买到国内工厂生产的，虽然也标注一样的型号，但是性能大不一样。例如，用国内的 si2306 和 si2307 后会出现烧管的问题。原因是耐压不够，特别是允许门极的电压幅度小。其解决方法是限制门极驱动电平。由图 3-28 可知，门极的电平幅度是 15V，为此，可在门极加入分压电阻。

图 3-29 是一种能适应国产和原装进口 MOS 管的比较稳定的 H 桥，在 Q2 的门极加入了分压电阻，因此门极的电压变化是 7.5V，这样对 MOS 管的指标要求就大大降低了，使其能采用国产的 MOS 管。

测试发现，有些国产的门极电压耐压偏低，国产的 MOS 价格是进口的一半不到。Q13 用于过流检测，当发生卡球故障，电动机过流时，通过 R29 取样，R17 和 C16 构成积分电路，用来对 R29 上的 PWM 电压进行滤波，形成相对平稳的电压，进入 Q13 并经放大后进入单片机。过流的灵敏度可通过调节 R29 或者调节 R17 和 C16 的大小来控制。时间常数的计算为 $t=0.70RC$ ，其中 t 要大于多个 PWM 周期。

图 3-28 中的 Q11 和 Q12 是驱动发球电动机的功率 MOS，由单片机直接驱动。R25 和 R26 是 MOS 管门极的限流电阻，MOS 门极具有电容，加了电阻可以起阻尼作用，防止振荡。R30 和 R31 是下拉电阻，单片机上电时引脚为高，电动机会突然转一下。另外，这个高电平的输出电流很弱，容易令 MOS 管处于放大状态，从而在上电瞬间损坏。加入下拉电阻可以防止出现这情况。D2、D3 是续流管，可在 MOS 关闭时使电动机电流不中断，避免线圈产生高压击穿 MOS 管。

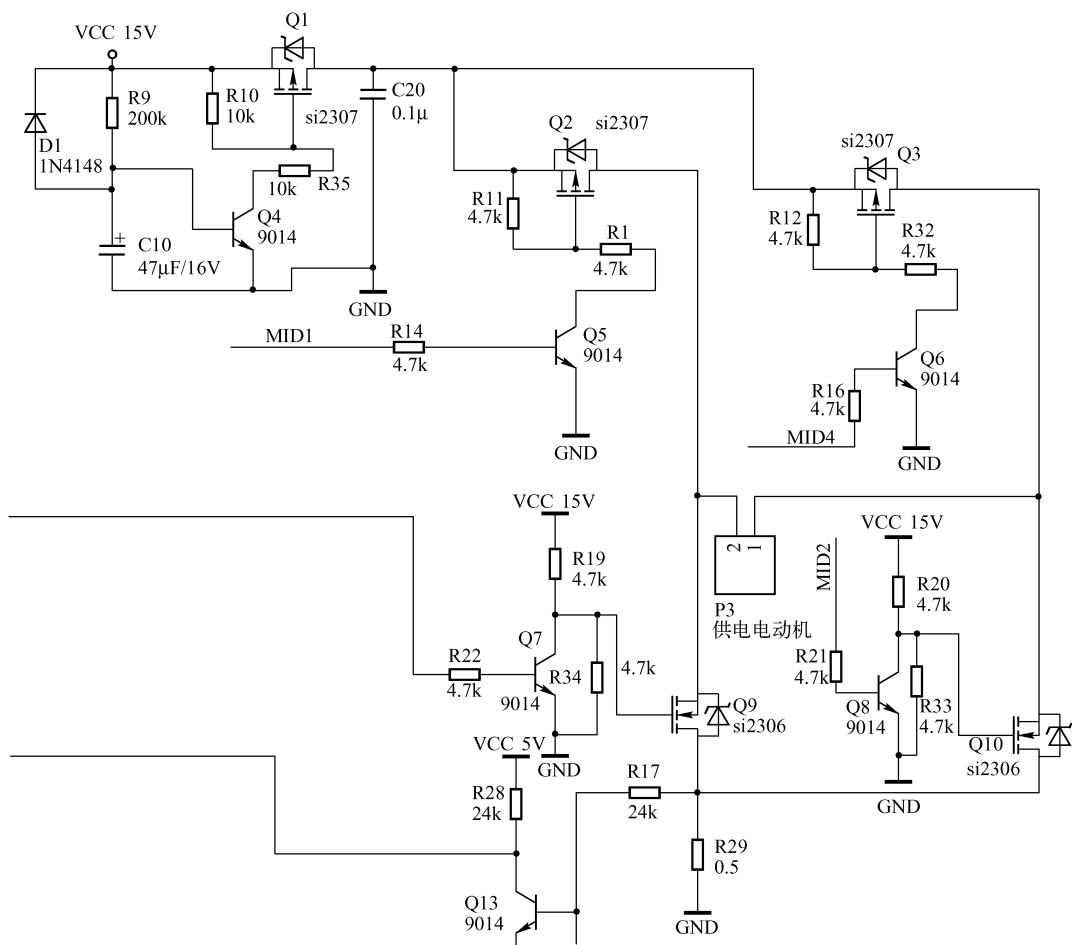


图 3-29 稳定的 H 桥

3.3 主控板硬件原理图设计

为了完成需求分析中提到的功能，主控电路板应该包括电源部分、语音提示部分、H 桥、舵机控制、CPU、供球电动机驱动和红外接收。图 3-30 是主板的完整原理图。

主板带语音提示功能，由 U2 和 U5 构成，U2 是 AP89085 语音芯片，U5 是 LM4871 功放。

AP89085 是一次性编程(OTP)语音芯片，采用 4-bit AD PCM 或 8-bit PCM 压缩方式。在 6kHz 采样率下时间长度可达 85s；通过 M0 和 M1 可以选择以按键或 CPU 方式触发，按键可以触发 32 段，CPU 可以触发 254 段；3 个输出端可以选择 LED、STOP、BUSY 的不同组合；声音输出可外接三极管放大输出(COUT)或直接采用推扬声器(VOUT)方式。AP89085 的工作电压为 2.6~3.6V，静态电流小于 5 μ A。芯片外围线路简单，语音片段需专用编程器烧录。AP89095 推荐的接线图如图 3-31 所示，按照芯片的说明书，与单片机之间的接口有并行和串行两种，为了节约 I/O 口，本设计可以采用串行接口方式。

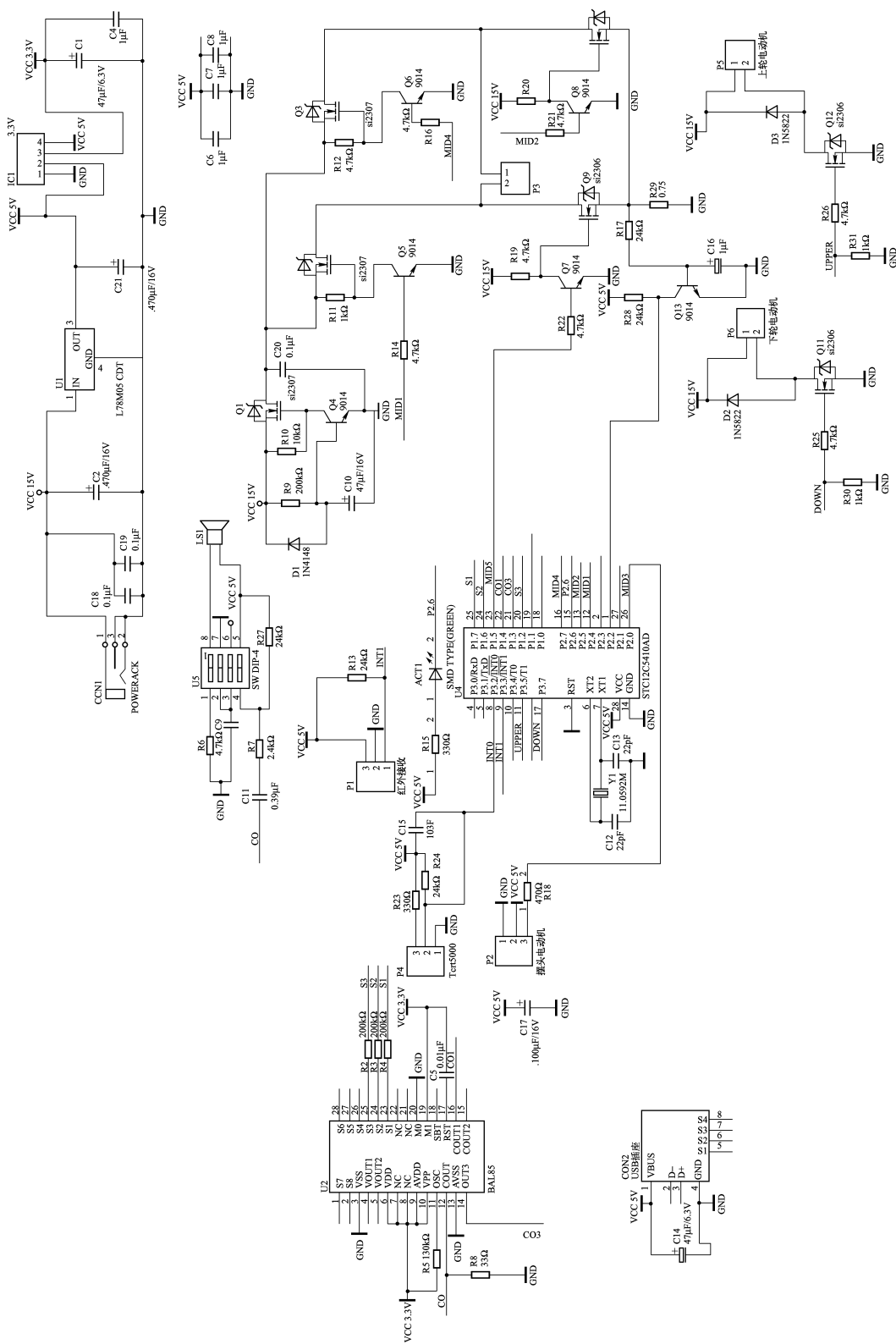


图 3-30 发球机控制板原理图

0x13	Seven
0x14	8
0x15	Eight
0x16	9
0x17	Nine
0x18	加
0x19	减
0x1A	Increase
0x1B	Decrease
0x1C	开始
0x1D	暂停
0x1E	Start
0x1F	Stop
0x20	时间到
0x21	Time out
0x22	自动卸球
0x23	unload balls
0x24	10
0x25	Ten
0x26	11
0x27	Eleven
0x28	取消
0x29	Cancel

图 3-30 的上方是电源部分，电源在本书前面有过介绍，这里不再重复。

P2 是舵机接口，单片机 I/O 直接把信号输入到舵机。

需要提到的是卡球检测。R29 是卡球检测电阻，发球机运行的时候有时会发生卡球，一旦卡球就要让供球电动机倒转，自动解决卡球的情况，进而恢复正常。发生卡球后电动机电流上升，导致 Q13 的基极电位上升。由于电动机运行时存在较大的电流波动，因此增加了滤波电容 C16，以防止卡球报警产生误动作。卡球信号是开关信号，通过三极管电平变换后最终进入单片机的 P2.2。

以上是发球机控制板的硬件设计，总结下来硬件分为以下几部分：

- (1) 电源系统；
- (2) 两个 PWM 电动机驱动；
- (3) 一个可变速的正反转供球电动机及过流检测；
- (4) 红外遥控接收部分；
- (5) 红外计数传感器部分；
- (6) 舵机控制部分；

- (7) 语音部分;
- (8) 卡球检测部分。

图 3-32 是主控板正反两面的样板照片, 样机为手工焊接, 等样机没有问题以后, 才能送贴片厂加工。主控板样机必须配合遥控器才能工作, 因此下面接着介绍遥控器的设计。

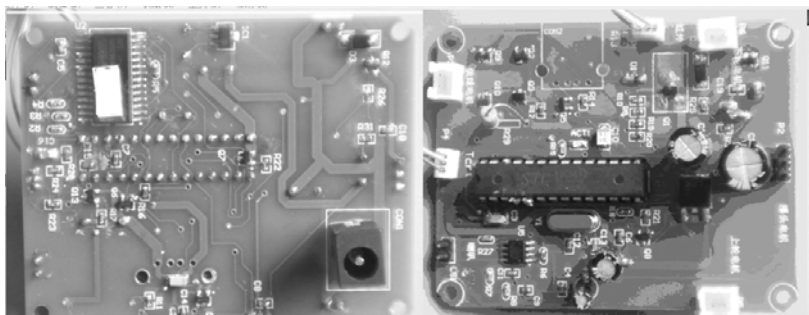


图 3-32 手工焊接的主控板样机照片

3.4 红外遥控发射硬件设计

遥控器的设计步骤为, 首先是遥控器硬件的设计, 然后是遥控器软件的设计。

3.4.1 遥控器硬件要求分析

首先, 遥控器使用的是电池, 用户不希望经常更换电池, 因此遥控器必须考虑功耗。其次, 为了在光线不是很亮的地方使用, 液晶要带背光。背光比较耗电, 也需要考虑功耗。遥控器带红外发射头, 同时还具有 5 个按钮, 参考图 3-3 和图 3-33。

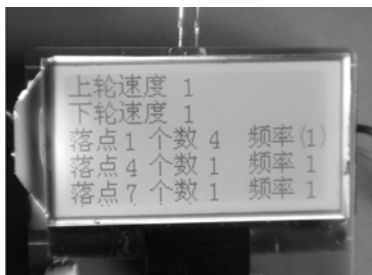


图 3-33 液晶的显示行数

3.4.2 液晶的选择

市场上有很多种类的液晶, 例如, 普通的液晶字符模块, 有内置中文和不带中文字库的两种, 前者的价格高于后者, 但程序书写方便, 后者需要字库支持。

图 3-34 中的液晶模块较多应用于工业场合, 用在本设备上并非合适, 因为本设备的遥控器要求轻巧, 为此选用比较薄的一种, 不带铁框和厚重的电路板。

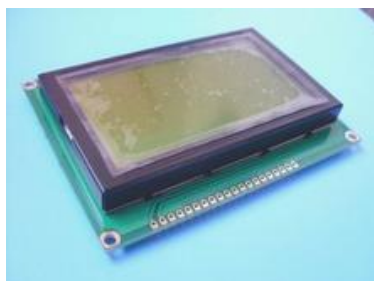


图 3-34 液晶字符模块

图 3-35 是按照设计要求选用的液晶, 根据这种液晶的外观找到相关的型号, 和本设计的要求相匹配。通过对比市场上的多种液晶, 决定采用 160×80 点阵不带字库的液晶。不带字库的液晶价格较低, 但对软件的编写要求高; 为了保存字模, 对 CPU Flash 容量也要求大一些。



图 3-35 适合本产品的液晶

采用标准汉字点阵, 每个汉字是 16×16 (单位为像素, 下同), 数字和英文点阵是 8×16, 一行可以显示的汉字数是 $80/16=5$ 。按照图 3-4 的要求, 液晶屏无法显示 17 行, 这样只能采用翻页的方法, 一共需要 4 页才能显示。液晶点阵如图 3-36 所示。

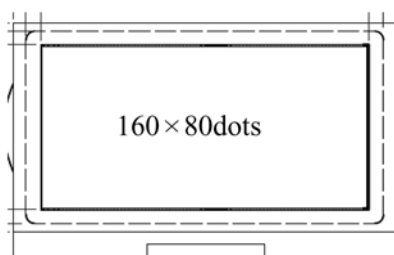


图 3-36 液晶点阵图

3.4.3 遥控器主板设计

遥控器主板主要由液晶、键盘和红外发射头组成。按照前面的需求分析, 设计的原理图如图 3-37 所示。

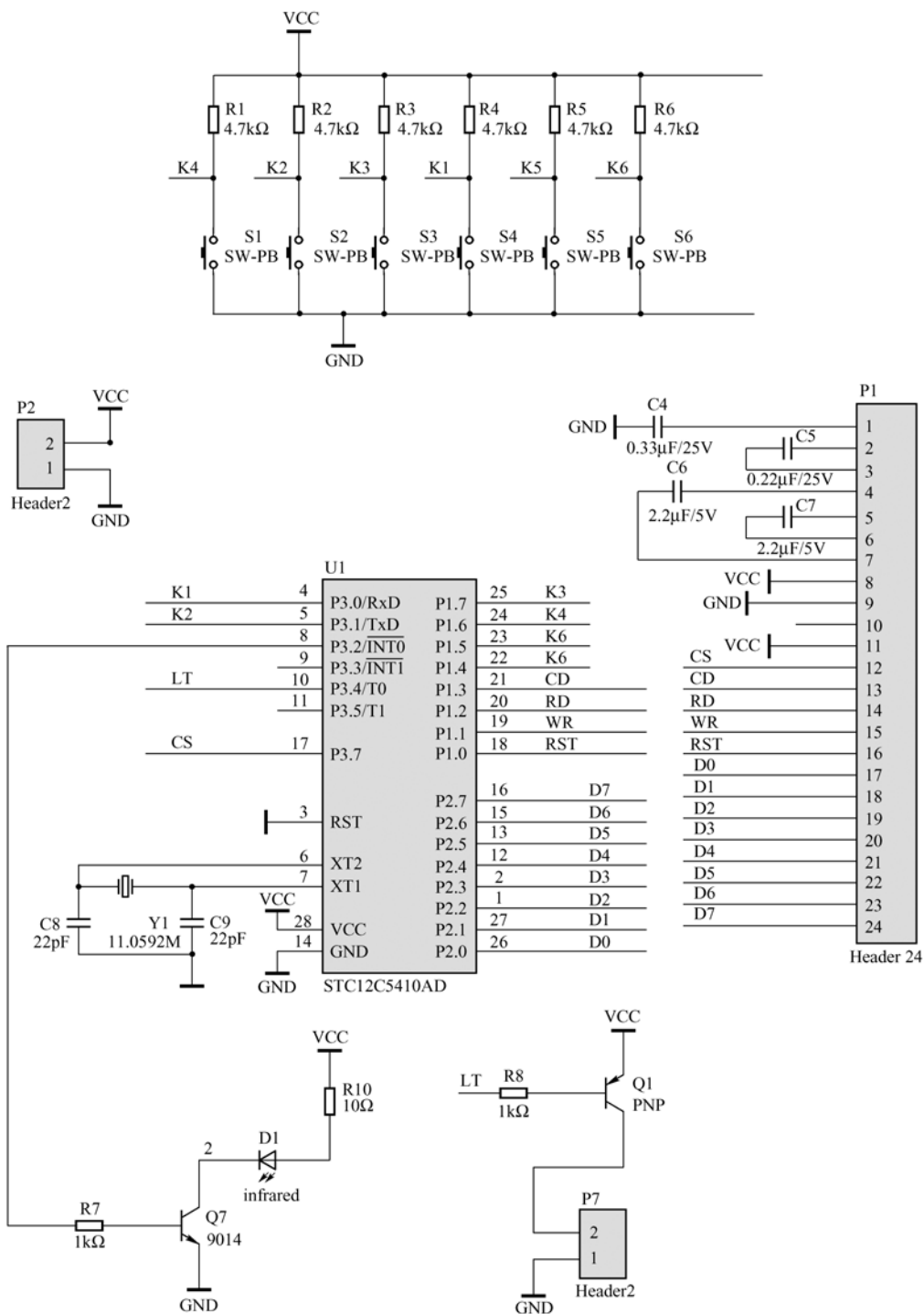


图 3-37 遥控器主板设计原理图

在原理图中，红外发射管发射瞬间的电流较大，采用三极管驱动。液晶有背光，通过

Q1 控制亮灭。当红外发射的时候，由于瞬间电流大，如果不注意布线，如红外发射的供电和液晶背光供电节点汇合处共同通过一段较长的印制线连接到电池端，因为走线电阻的影响则电流大时在这段走线上有较大的压降，会导致电压波动，产生背光闪烁。因此必须将红外发射的电源 R10 的上方直接连接到电池供电的输出端。

3.4.4 遥控器红外发射的调制

由图 3-37 可以知道，线路包括液晶接口和红外发射两部分。红外发射管采用 NPN 三级管驱动。红外发射距离与发射管流过的电流成比例关系，但电流达到一定数值以后，电流上升对距离的影响变化就很小了。为了节省发射功耗，采用常用的减小占空比的方法。减小占空比能降低发射功耗，提高瞬间的发射功率。从图 3-37 中还可知道，减小 R10 的阻值相当于增大集电极电流，但如果持续大电流流过发射管，会导致发射管提前老化报废。

图 3-38 显示的是两个占空比不同的红外发射波形，第一个占空比为 100%，第二个占空比为 20%；第一个增加发射电路会导致发射管发射效率低下，电池消耗明显加大，第二个由于平均功耗小，因此发射效率较高。

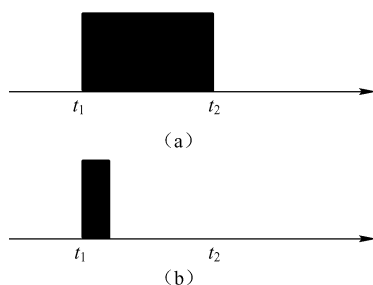


图 3-38 红外发射占空比

红外发射采用的是 38kHz 的调制波，图 3-39 是红外发射头的输出波形。

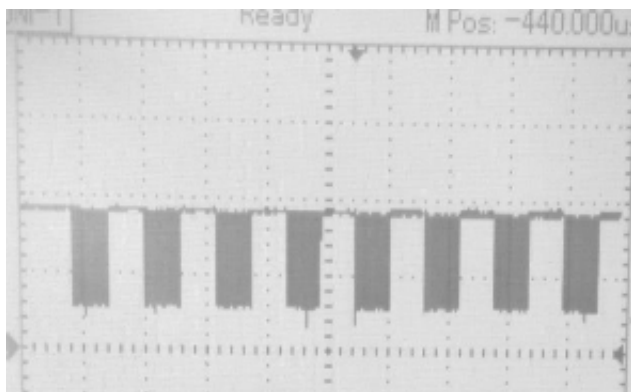


图 3-39 红外发射头的输出波形

调制波的频率很高，因此在示波器上需要将时间轴放大，形成如图 3-40 所示的情况。

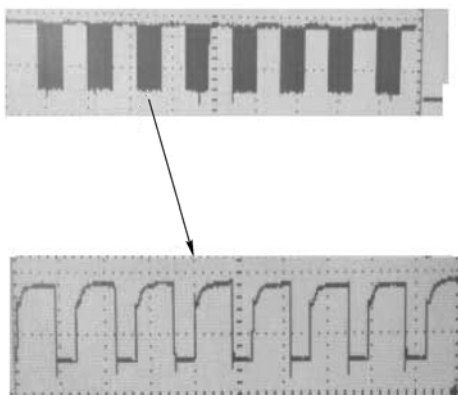


图 3-40 红外调制波

图 3-40 中每个深色的区域其实就是 38kHz 的载波，载波的占空比约为 1/3。
图 3-41 是解调并去掉载波后的波形，单片机将对此信号进行解析，获得遥控接收的数据。

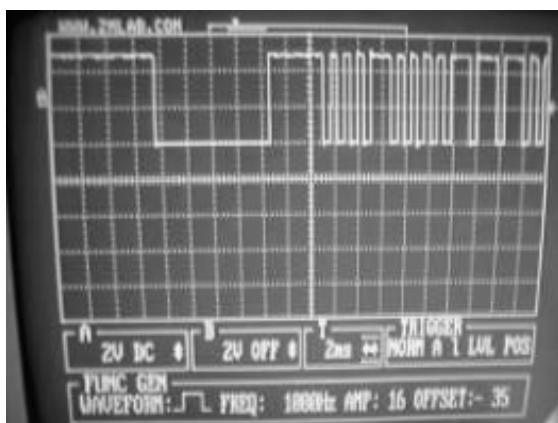


图 3-41 红外接收解码波形

3.4.5 遥控器的外观

遥控器电路板如图 3-42 所示。

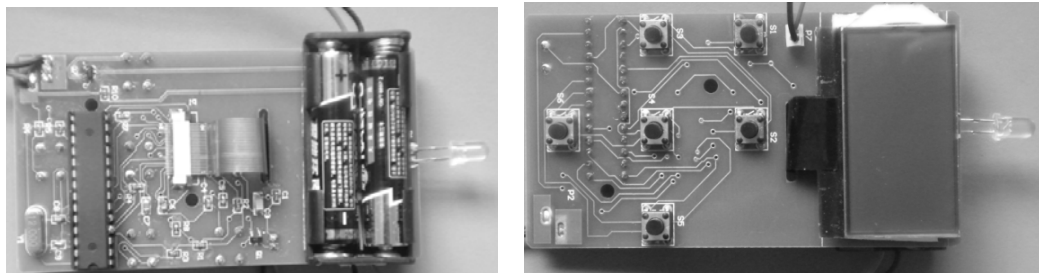


图 3-42 遥控器电路板

4 个按钮对应图 3-3 中遥控器的按键布局，总共由两节电池供电，单片机本身具有休眠功能，整机功耗很低。按中间的按钮能将单片机从休眠中唤醒。

图 3-43 是软件运行后的遥控器显示器。

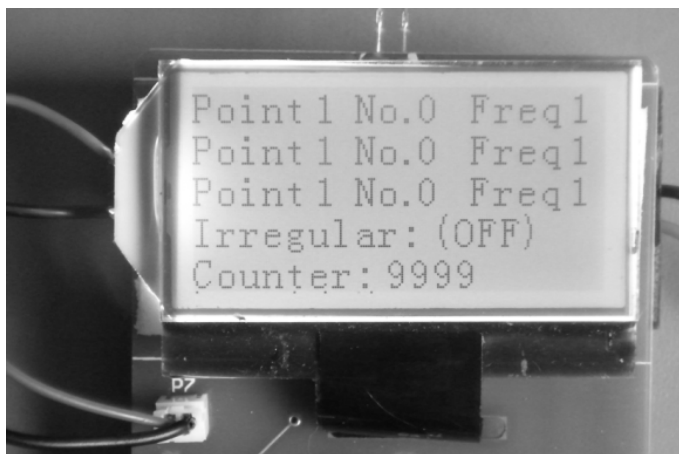


图 3-43 软件运行后的遥控器显示器

3.5 软件设计规划

下面介绍软件的设计。

软件的设计分为两部分，分别是控制主板的软件设计和遥控器的软件设计。控制主板的软件设计看上去不复杂，但实际上并不容易。

3.5.1 合理安排中断优先级

由于 51 单片机资源很少，因此如何合理使用资源，满足实时的要求，成为最关键的问题。最主要的部分是舵机的控制，其对实时要求很高，不能有任何延迟，否则舵机会抖动。另外就是红外接收部分，红外接收也不能有延迟，否则接收不正确。因此在中断程序中哪个先处理、哪个后处理非常关键。还有就是，供球电动机边上有一个光电计数传感器，用来计算供球的数量，软件也不能有延迟。

从时间的要求上看，把舵机的中断优先级设置为最高，能保证舵机不发生抖动，同时由于舵机的中断处理时间很短，因此不会影响红外接收中断的处理。如果舵机的中断处理时间很长，则一定会影响红外的接收，这样就不能采用本方案，必须在硬件上重新进行处理。

舵机的控制采用定时器中断，每 $50\mu\text{s}$ 进入一次中断，在中断内对输出的脉冲宽度进行控制，用软件的方式实现脉冲宽度的变化。其中 MID3 指的是舵机控制信号。

```
void time0(void) interrupt 1 using 2 //定时中断
{ //will be entered every 50μs
```

```
static unsigned int k=0;

//202ms
//20.2ms
//k is 5 to 25
if(k==dj_final) MID3=0;

if(k++==400){//20ms
MID3=1;
    if(pause==1) {mytime++;dingshiqi++;} //will be added every 20ms
k=0;

}

}
```

红外遥控接收的控制与第 2 章内容类似，红外的接收首先是判断起始信号，然后在中断内读取当前持续的时间，一直等到红外信号全部处理完毕才推出中断。

读者可能有疑问，这样不是影响舵机的实时性了吗？其实不会，这是因为舵机的中断优先级最高，当舵机的中断到来后，外部中断立刻被打断，开始处理舵机的中断。

优先级在 51 单片机内是固定的，但这里用了 STC 单片机，因此采用关键词 `using x` 来重新确定中断优先级。

以下红外中断和第 2 章的内容非常相似，具体解释在第 2 章已经说明。

```
void INT1_Routine(void) interrupt 2 using 1
{

char i,j,tmp;

bit err=0;
TH1=0;
TL1=0;

while((tst=infrared)!=1) {if((TH1>15)|| (err==1)) {err=1;return;}};
if(TH1>0xa)
    {TH1=0;
    TL1=0;}
else return;

while((tst=infrared)!=0) {if((TH1>5)|| (err==1)) {err=1;return;}};
```

```

    if(TH1>3)
        {TH1=0;
        TL1=0;

        }
    else return;

    index=-1;

    for(i=0;i<42;i++)
    {
        tmp=0;
        for(j=7;j>=0;j--)
        {
            TH1=0;TL1=0;
            while((tst=infrared)!=1) {if((TH1>4)||(err==1)) {err=1;return;}}
            TH1=0;TL1=0;
            while((tst=infrared)!=0) {if((TH1>4)||(err==1)) {err=1;return;}}

            if(TH1>=2)          tmp|=(0x1<<j);

        }

        //-----
        if(err==0) {mdata[i]=tmp;index=i;}

        if((i==5)&&((mdata[4]>>7)==0)) {
            if(mdata[0]!=0){rcvsuccess=1;return;}
            else index=-1;}

    }

    rcvsuccess=1;

}

```

3.5.2 主控程序总体结构

主控程序需要对红外遥控器发来的命令进行处理，根据发球规则调整舵机，以及供球电动机、发球电动机的速度，判断当前发球的个数是否达到预设的数量，判断发球的时间是否达到设置的时间，并判断是否有卡球情况发生，如果发生就控制倒转。主控程序流程图如图 3-44 所示。

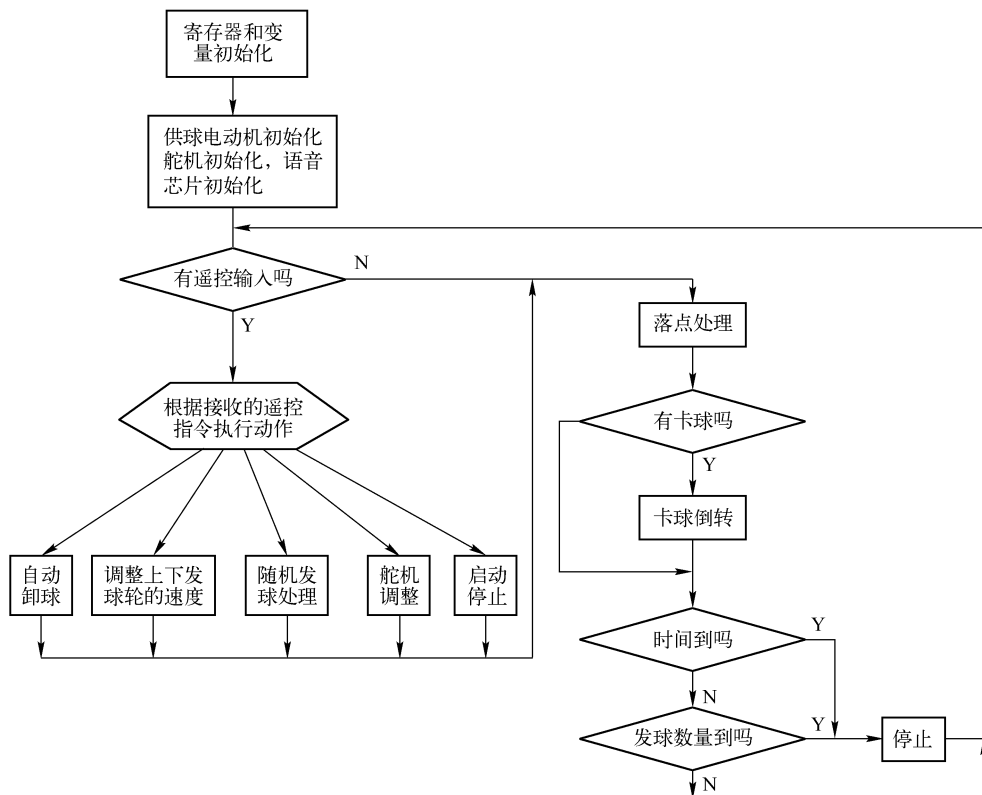


图 3-44 主控程序流程图

主控程序如下：

```

main()
{

xdata unsigned int i;
unsigned char j;
char dj=4;
bit myld=0;
char varindex=6;
unsigned char xdata localtimesetting=0;
PT0=1;

P2M0=0x06;
P2M1=0xf9;

P3M0=0x0c;
P3M1=0xf0;
    
```

```

P1M0=0x18;
P1M1=0xe4;

MID4=0;
MID2=1;
MID5=1;

TMOD=0x12;

CCON=0;
CL=0;
CMOD=0x2;
CCAP0H=CCAP0L=lower_percent; //下轮速度
CCAPM0=0x42;

CCAP1H=CCAP1L=upper_percent; //上轮速度
if(upper_percent==255) PCA_PWM1=0x3;if(lower_percent==255) PCA_PWM0=0x3;
CCAPM1=0x42;

dj_final=CST; //舵机初始位置设置

IE0=0;
CR=1;
TR0=1;
IT0=1;//external interrupt config falling
TH0=0x0; //50μs ;
TL0=0x0;
TR1=1;
ET0=1;
EX1=1;
EA=1;
CS=0;
SCK=0;
CCAPM3=0x42;
CCAP3H=CCAP3L=bottom_PWM_fixspeed; //初始化供球电动机的速度
PCA_PWM3=0x0;

bottom_run_stop=0; //启动时供球电动机不转
ball_number=2;

mytime=0;
dingshiqi=0;
rcv=0;
for(j=0;j<42;j++)

```

```

mdata[j]=0;
CS=1; //片选语音芯片
write_comd(PUP2); //打开语音芯片
CS=0; //关闭语音芯片
LED=1;
longms(1);

dj_run(dj); //初始化舵机，让舵机指向中间

while(1)
{
//以下为查询接收到的遥控指令
if(rcvsuccess) //如果接收到的遥控指令没有错误，就做进一步处理，否则跳过
{
rcvsuccess=0; //接收成功标志清零
if((mdata[5]&0x1)==0) overtime=0;
index=-1;
if(index>=41) {myld=0;varindex=6;}
if(((mdata[5]>>2)==0x22)||((mdata[5]>>2)==0x23)) ; //自动卸球
else
voice(mdata[5]>>2);
if(((mdata[5]&0x2)!=0)&&(pause==1)){ mdata[5]&=0xfd;}
if((pause==1)&&(overtime==0)) mdata[5]&=0x3;

if(mdata[3]!=localtimesetting)
{
dingshiqi=0;
localtimesetting=mdata[3];
}

if(totalball!=(unsigned long)((unsigned long)mdata[1]<<8|mdata[2]))
totalballtemp=totalball=(unsigned long)((unsigned long)mdata[1]<<8|mdata[2]);

if(((mdata[5]&0x1)==1)&&(overtime==0)&&(pause!=2))
{

switch(mdata[0]>>4) //上轮速度
{
case 0: upper_percent=255; break;
case 1: upper_percent=160; break;
case 2: upper_percent=140; break;
case 3: upper_percent=120; break;
case 4: upper_percent=100; break;
case 5: upper_percent=87; break;

```

```

case 6: upper_percent=69; break;
case 7: upper_percent=51; break;
case 8: upper_percent=42; break;
case 9: upper_percent=30; break;
    }

    switch(mdata[0]&0x0f)//下轮速度
    {
case 0: lower_percent=255; break;
case 1: lower_percent=181; break;
case 2: lower_percent=175; break;
case 3: lower_percent=162; break;
case 4: lower_percent=154; break;
case 5: lower_percent=147; break;
case 6: lower_percent=136; break;
case 7: lower_percent=121; break;
case 8: lower_percent=111; break;
case 9: lower_percent=99; break;
    }

    if(upper_percent==255) PCA_PWM1=0x3; else PCA_PWM1=0;

    if(lower_percent==255) PCA_PWM0=0x3; else PCA_PWM0=0;

    CCAP0H=CCAP0L=lower_percent; //lower motor
    CCAP1H=CCAP1L=upper_percent; //upper motor
    CCAP3H=CCAP3L=bottom_PWM_fixspeed; //mid1
    if(((mdata[5]&0x1)==1)&&(pause==0)) {longms(2); pause=1;}//wzxwzx
    }

    else if((mdata[5]&0x1)==0)
    {
        totalballtemp=totalball=(unsigned long)((unsigned long)mdata[1]<<8|mdata[2]);
        pause=0;
        dingshiqi=0;
        myld=0;
        varindex=6;

        }

    }

    if(((mdata[5]&0x1)==1)&&(myld==0)&&(overtime==0)&&(pause!=2))
    //判断当前的落点数据是否处理完毕

```



```

{

    dj=mdata[varindex++];
    ball_number=mdata[varindex++];
    i=mdata[varindex++];

    if((mdata[5]&0x2)!=0)    //无规律发球允许
    {
        srand(TL0);        //初始化随机数发生器
        dj=rand()%8;        //计算出舵机当前的落点
        ball_number=rand()%10;    //计算出发球个数
        if(ball_number==0) ball_number=1;
        i=rand()%10;
        //ball_number
    }

    if(varindex>=38) varindex=6;

    if(ball_number!=0) //供球电动机的速度
    {
        switch(i) //根据随机数发生器获得的供球电动机速度
        {
            case 1:back_percent=128;break;//20
            case 2:back_percent=78;break; //30
            case 3:back_percent=53;break; //40
            case 4:back_percent=38;break; //50
            case 5:back_percent=28;break; //60
            case 6:back_percent=20;break; //70
            case 7:back_percent=15;break; //80
            case 8:back_percent=11;break; //90
            case 9:back_percent=8;break; //back_percent=3000/n -22  n 是一个数
            default: back_percent=28;
        }
        bottom_pause();//供球电动机暂停
    }
}
/*
参照图 3-10 舵机转角和输入脉冲的关系，将舵机角度分割为 11 个指向
0.5ms-----0° ;
1.0ms-----45° ;
1.5ms-----90° ;
2.0ms-----135° ;
2.5ms-----180° ;
*/
//dj 为 1~11

```

```

        dj_run(dj); //舵机执行，指向规定的落点
    }
    myld=1;
}

if(((mdata[5]&0x1)==1)&&(overtime==0)){
    if(bottom_run_stop==0){
        mytime=0;
        bottom_run_stop=1;
    }
    if((mytime>=back_percent)&&(pause==1)) foward_backward(F); //每转一下的停止时间
back_percentx20ms

}

if(mytime>10)
    detectKQ(); //卡球检测，万一卡住，则供球电动机必须倒转
//自动卸球

if(((mdata[4]>>4)==1)&&(pause==0)){
    mdata[4]&=0x0f;
    ball_number=8; //自动卸球个数
    EX1=0;
    voice(mdata[5]>>2);
    stop();
    longms(2);
    while(ball_number!=0)
    {
        if(IE0)
        {
            IE0=0;
            if(ball_number!=0) ball_number--; //卸除指定数量的球
            if(ball_number==0) {myld=0;}
        }

        foward_backward(B); //供球电动机倒转执行
    }
    EX1=1;
    MID4=0;
    MID2=1;
}

if((localtimesetting!=0)&&((mdata[5]&0x1)==1)&&(overtime==0))
if(gettime()>=localtimesetting){ //规定的运行时间到

```

```

localtimesetting=0;
stop();//机器停止
myld=0;
for(;;){
if((mdata[4]&0x0f)==3) voice(0x21);//中文语音播放时间到
else voice(0x20);//英文语音播放时间到
overtime=1;
if(index!=-1) break;
longms(1);
i
}
}

LED=sensor?1:0;
//如果接收到暂停遥控指令，则所有电动机停止运行
if(pause==0) {PCA_PWM1=0x3;PCA_PWM0=0x3;
                CCAP0H=CCAP0L=0xff;
                CCAP1H=CCAP1L=0xff;
                bottom_pause();
            }
    if(ball_number==0) myld=0;
else if(IE0)
    {
        //
        IE0=0;
        for(i=0;i<1000;i++);
        if(TCRT5000==0){
            //每成功供球一个，供球电动机停止一个节拍
            mytime=0;
            bottom_pause();

            IE0=0;
            if(totalball>0)
                if(--totalballtemp==0) //每供球一个，数量递减，直到数量为零，停止运转
                {
                    stop();
                    myld=0;
                    pause=2;
                }

            if(ball_number!=0) ball_number--;
            if(pause==0) stop();

        }
    }
}

```

```
}  
}
```

至此，桌面式乒乓自动发球机的设计介绍完毕。在用单片机设计了两个产品以后，下面要转到近年来另外一款主流的嵌入式 CPU——ARM 处理器。ARM 的价格便宜，内部性能优于单片机，并且功能强大，能跑操作系统，在电子设备上获得了广泛的应用。对于用惯了单片机的人而言，要学习 ARM 并不是一件很容易的事情，因此需要有一个平稳的过渡过程。

第 4 章

质的飞跃——从单片机到 ARM 产品开发

单片机资源有限，而最近十多年出现的 ARM 大大挤压了单片机的生存空间，ARM 处理器无论在资源还是成本上都远远超越了单片机。本章首先介绍 ARM 相关的基础知识，然后循序渐进、从没有操作系统的 ARM 开发介绍至带操作系统的开发，包括文件系统、图形系统及嵌入式 Web 服务器的开发。

4.1 嵌入式系统和 ARM

嵌入式系统这个概念其实并不是很新，它伴随着微处理器的发展已经有多年的历史。那么，为什么直到最近几年嵌入式这个名词才出现呢？这是由于最近几年，随着信息技术的发展，许多控制器的功能越来越复杂，原来可以用数字逻辑电路或者模拟电路实现的场合现在考虑到性能及体积都采用微控制器来实现，而这些控制器就嵌入在设备中，这些设备的数量越来越多。因此，嵌入式一词这几年也被炒得火热。

目前，嵌入式系统的定义有许多种，常见的定义是：嵌入式系统是一种对资源有严格限制的、软件和硬件紧密耦合并且与外界环境存在交互的系统。嵌入式是指计算机，它是某个组成模块的一部分，对用户不是直接可见的。由于应用场合的多样性，嵌入式系统通常和实时性有密切关系，也就是在嵌入式硬件上运行实时软件。实时软件对时间的要求很严格，系统运行结果的正确性不但和逻辑结果有关，而且还与给出结果的时间有关。如果超过规定的时间输出结果，运行就是不正确的。当然，实时软件也可以运行在台式机上，不过这台计算机最好只运行这一个任务。

ARM (Advanced RISC Machines)，既可以认为是一个公司的名字，也可以认为是对一类微处理器的通称，还可以认为是一种技术的名字。

1991 年 ARM 公司成立于英国剑桥，主要出售芯片设计技术的授权。目前，采用 ARM 技术知识产权 (IP) 核的微处理器，即通常所说的 ARM 微处理器，已遍及工业控制、消费类电子产品、通信系统、网络系统、无线系统等各类产品市场，基于 ARM 技术的微处理器应用约占据 32 位 RISC 微处理器 75% 以上的市场份额，ARM 技术正在逐步渗入到我们生活的各个方面。

ARM 公司是专门从事基于 RISC 技术芯片设计开发的公司，作为知识产权供应商，其本身不直接从事芯片生产，靠转让设计许可由合作公司生产各具特色的芯片。世界各大半导体生产商从 ARM 公司购买其设计的 ARM 微处理器核，根据各自不同的应用领域，加入适当的外围电路，从而形成自己的 ARM 微处理器芯片投入市场。目前，全世界有几十家大的半导体公司都使用 ARM 公司的授权，因此既使得 ARM 技术获得了更多的第三方工具、制造、软件的支持，又使整个系统成本得以降低，使产品更容易进入市场被消费者所接受，更具有竞争力。

当前，许多嵌入式系统都采用 ARM 芯片，ARM 是一种基于 RISC 架构的低功耗、高性能微处理器核。许多著名的半导体厂商从 ARM 公司购买 ARM 核并加入各自的外围电路，形成自己的芯片。ARM 和单片机相比具有很多相似之处。例如，都在片内集成外设，CPU 自身工作都不需要加外围控制器，都是小体积。因此，ARM 完全可以代替单片机，当然成本可能有些偏高，有点大马拉小车的味道。但是 ARM 具有单片机无法比拟的优点，例如，存在 MMU 和不同的异常，从而为运行嵌入式操作系统提供了条件；存在存储器加速模块、cache 及多级流水线，从而使高性能获得保障，使运行诸如网络协议、USB 协议及 CAN 总线协议等成为可能。

开发嵌入式系统不是一件容易的事情，困难主要表现在三个方面。首先，它是软、硬件密切配合的紧密耦合系统，各种通信协议也比较复杂；其次，由于市场竞争压力，如果产品不能在一定的时间里出来，将失去开发的意义；再次，由于工作环境的复杂性，系统测试比较困难，维修的代价也比较高。

4.1.1 JTAG 接口

首先需要弄明白的是，为什么现代的微处理器普遍采用 JTAG？这是因为当前许多复杂的微处理器的内核不能再通过芯片的外设直接访问，调试芯片程序变得困难；另外，为了缩短开发周期、加快产品进入市场的速度，直接将 CPU 安装在电路板上调试也更接近实际工作情况。这些都需要 JTAG 装置来完成。

4.1.2 JTAG 标准

JTAG 是 The Joint Test Group 的缩写，它是由一些主要的电子生产企业在 1985 年创立的印制电路板和集成电路测试标准。JTAG 标准最终在 1990 年被 IEEE 接受，成为 IEEE 1149.1 Test Access Port and Boundary-Scan Architecture（访问端口测试和边界扫描体系），该标准定义了边界测试所应具备的硬件和软件条件。

包含 JTAG 的器件可以是微处理器、微控制器、PLD、CPLD、FPGA、ASIC 或者其他遵循 1149.1 规范的数字处理器。

1. JTAG 的引脚定义

遵循 JTA 规范的器件包含以下几个引脚。

TCK：测试时钟输入，它和系统时钟不同。

TDI：测试数据输入，通过它数据移位进入器件。

TDO：测试数据输出，通过它数据从器件移出。

TMS：测试模式选择，在 JTAG 规范中 TMS 命令用于选择测试模式。

TRST：测试复位输入，它为 TAP 控制器提供异步初始化。

器件的测试支持功能是通过 TAP 控制器实现的。TAP 是一个状态机，它控制所有相关操作，每种遵循 JTAG 的器件都有自己的 TAP 控制器，通过 TCK 和 TMS 可以使状态机内部的状态发生变化，从而支持诸如断点、单步、内部观察等调试工作。

2. 边界扫描寄存器

IEEE 1149.1 规范约定移位寄存器的一个单元在设计时加入 IC 逻辑，并且和 IC 的每个数字引脚连接，此单元称为边界扫描单元，将 JTAG 电路和 IC 内部逻辑连接。IC 的所有边界扫描单元组成边界扫描寄存器。当执行 JTAG 测试时，边界扫描单元逻辑被激活；在 IC 正常工作的时候，它处于不活跃状态。边界扫描单元逻辑的连接构成了扫描链。

JTAG 的内部结构如图 4-1 所示。

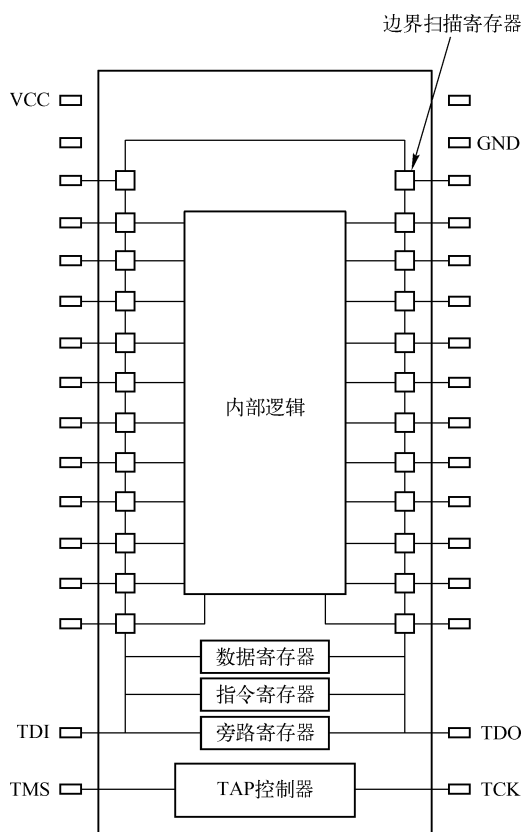


图 4-1 JTAG 的内部结构

4.1.3 JTAG 硬件控制器

具备 JTAG 功能的器件和 PC 之间的联系是通过硬件控制器实现的，硬件控制器一端连接 PC，另一端连接遵循 JTAG 标准的器件，如图 4-2 所示。

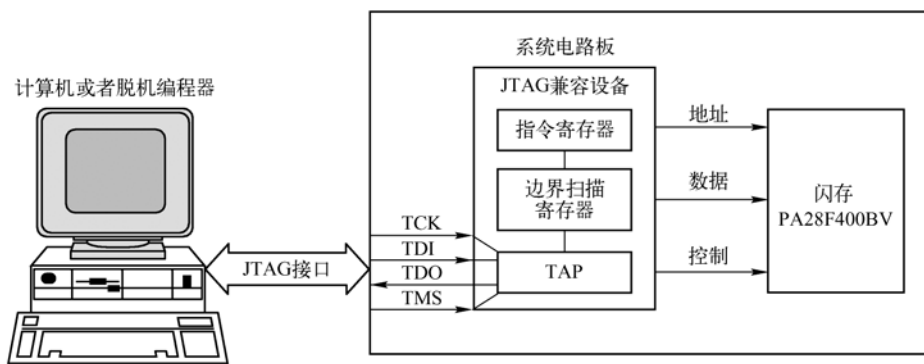


图 4-2 JTAG 调试仿真

JTAG 硬件控制器和 JTAG 器件通过 TAP 控制器通信，通过 TCK 及 TMS 输入状态机命

令（如断点、单步）。操作数据的三个 TAP 指令如下。

- **SAMPLE/PRELOAD**: 该指令用来将边界扫描单元的数据读出或者把数据送入边界扫描单元。
- **EXTEST**: 该指令用来使连接器件输入引脚的边界扫描单元的功能变为输入引脚的状态信号检测，而和输出连接的边界扫描单元则把数据传入互连的器件，如图 4-2 中的 Flash Memory 就是和 JTAG 器件互连的器件。
- **BYPASS**: 用来将边界扫描单元的移位数量降低到一个。例如，假设一个 CPU 包含 101 个边界扫描单元，那么 BYPASS 执行后边界扫描单元就只有一个。

4.2 JTAG 仿真器制作

4.2.1 ARM 的调试结构

本节针对 ARM 7 介绍调试结构。ARM 的调试体系采用协议转换器使调试器通过 JTAG 与 ARM 核直接通信。前面 JTAG 标准中提到的扫描链的功能是测试，这里将它用于调试——捕获数据总线上的信号并向内核或存储器插入新的信息。ARM 7 TDMI-S 核内具有 Embedded ICE 逻辑，Embedded-ICE 逻辑提供对片内调试的支持。调试指令直接通过扫描链插入 ARM 内核并执行。根据插入调试指令的不同，内核可以处于观察、保存或改变状态。ARM 的调试体系可以使程序指令执行速度处于调试速度或全速运行。在 ARM 中采用 JTAG 的特点是：通过 JTAG 接口可以观察 ARM 内核状态和系统状态（注意，系统状态包括片内外设，不同于内核状态）；不占用额外的目标系统资源；提供传统的断点访问和观察点访问；不再需要另外的 UART 端口来和监控程序通信。

围绕 ARM 内核有两个扫描链：围绕整个内核外围的一个扫描链与仅仅覆盖数据总线和断点的扫描链。由于后者的链比较短，从而使调试指令和数据可以快速插入内核，避免了额外的时间。

4.2.2 JTAG 仿真环境

如图 4-3 所示，ARM 的 JTAG 调试需要如下设备：

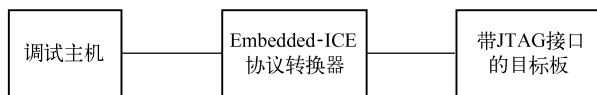


图 4-3 JTAG 调试环境

- 一台运行调试软件的主机。
- 一个 Embedded-ICE 协议转换器。Embedded-ICE 协议转换器将远程调试协议命令转换成所需要的 JTAG 数据，从而对目标系统上的 ARM 7 TDMI-S 内核进行访问。它包括两部分：将串行数据信号转换成 JTAG 接口兼容信号的装置和带有 JTAG 接口的 ARM 调试体系内核。

其中，前一部分可以是仿真器硬件，后一部分是 ARM 本身就支持的。

根据功能需要，仿真器硬件可以做得比较复杂也可以很简单。本书将教你如何自己做简单的仿真器。

调试主机运行调试程序，如 ADS、RealView、SDT 等。ICE 协议转换器其实包含两部分：协议转换硬件和软件。复杂的硬件通常采用 FPGA 实现，简单的可以用一片数字缓冲电路实现。协议转换软件一般在调试主机上运行，可以是 API 的方式也可以用后台服务的方式。调试主机和协议转换硬件之间可以采用各种方式连接：并行、串行、网络及 USB。只要最终递交给 ICE 协议转换器的数据一致就可以，这是通过运行在 ICE 协议转换器上的通信软件实现的。ICE 协议转换器和目标板通过遵循 JTAG 电气规范的电缆连接。表 4-1 所示为遵循 JTAG 规范的器件引脚描述。

表 4-1 遵循 JTAG 规范的器件引脚描述

引脚名称	类型	描述
TMS	输入	测试模式选择。TMS 引脚选择 TAP 状态机中的下一个状态
TCK	输入	测试时钟。该引脚允许 TMS 和 TDI 引脚上数据的转换。它是一个上升沿触发时钟，由 TMS 和 TCK 信号定义器件的内部状态
TDI	输入	测试数据输入。移位寄存器的串行数据输入端
TDO	输出	测试数据输出。移位寄存器的串行数据输出端。器件中的数据在 TCK 信号的下降沿输出
nTRST	输入	测试复位。NTRST 引脚可用于复位 Embedded-ICE 逻辑中的测试逻辑
RTCK	输出	返回的测试时钟，叠加到 JTAG 端口的额外信号。基于 ARM 7 TDMI-S 处理器内核进行设计时需要该信号。Multi-ICE（ARM 的开发系统）使用该信号来保持与低或宽范围时钟频率的目标系统的同步

4.2.3 自制简易仿真器

我们要做的仿真器一头接计算机并口，一头接 JTAG 口。通过 Wiggler 软件实现 PC 并口协议到串行 JTAG 协议的转换，利用高速 JTAG 串行扫描链，通过调试通信通道 Debug Communications Channel (DCC)连接 ARM 核心内嵌的名为“Embedded-ICE”的调试逻辑，调试逻辑实时监测 ARM 核心的寄存器、数据总线和地址总线。调试器设置 Breakpoint 及 Watchpoint 后，程序在 ARM 内核全速运行，调试逻辑实时监测地址和数据总线并与预设值比较，在吻合时产生异常中断通知内核并把控制权交给调试器。这样，在程序全速运行时，可以在断点处停止，可以设置条件断点、条件观测断点等，而不占用 CPU 时间及内存资源。目前的仿真器支持两个断点。

4.2.4 JTAG 仿真器硬件制作

目前市场上的 ARM 仿真器价格高昂，为了使读者能够在不花钱的前提下自己开发 ARM 产品，本节将介绍如何自己动手制作仿真器。从 4.2.2 节的图 4-3 可以发现，调试系统硬件包括调试主机、协议转换器及目标板。调试主机可以是运行 Windows 的 PC，ICE 协议转换器自己做，目标板是要开发的产品，也必须自己做。调试系统的软件包括 ADS 1.2、协议转换软件。其中协议转换软件比较复杂，它处理和 ADS 之间的接口及 JTAG 信号的转换，读者自行开发不现实，可从网上下载现成的软件。因此，读者要做的就是一件事：自制

仿真器硬件。最简单的仿真器可以是 Wiggler 电缆，Wiggler 电缆一端通过并口和调试主机连接，另一端通过 JTAG 接口连接目标板，如图 4-4 所示，其上方是并口，下方是 JTAG 接口。

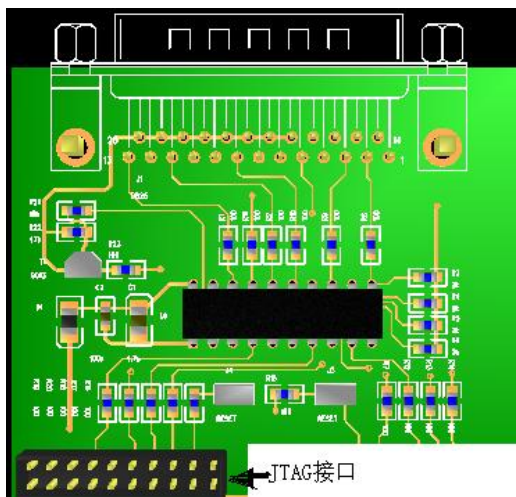


图 4-4 Wiggler 电缆

通用 ARM 简易仿真器可以适应各种类型的 ARM CPU，配合本书光盘提供的调试代理软件，就可以在 ADS 环境下调试。

目前，我们提供的简易仿真器支持两个断点，特别适合学习用。功能包括：

- 单步运行；
- 全速运行；
- 变量查看、修改；
- 寄存器查看、修改；
- 运行到光标停止；
- 运行到断点停止；
- 单步进入；
- 单步跳出；
- 当前上下文运行局部变量查看、修改；
- 当前上下文运行全局变量查看、修改；
- 内存数据查看、修改；
- 符号查看；
- 支持 ARM SDT 2.50、ARM SDT 2.51、ARM ADS 1.1、Arm ADS 1.2。

图 4-5 是 JTAG 的原理图，该电路的作用是把调试代理在计算机并口发出的信号转换为符合 JTAG 规范的信号，图中并口和 JTAG 端口之间遵守 Wiggler 电缆的定义。还有一种是 Predefined 电缆定义，它们之间存在细微差别。当然，用户也可以自己定义，但这样就必须修改调试代理软件。Wiggler 不需要外接电源，由于其消耗的功率非常低，因此只要连在目标板上，借用目标板的电源就可以了。

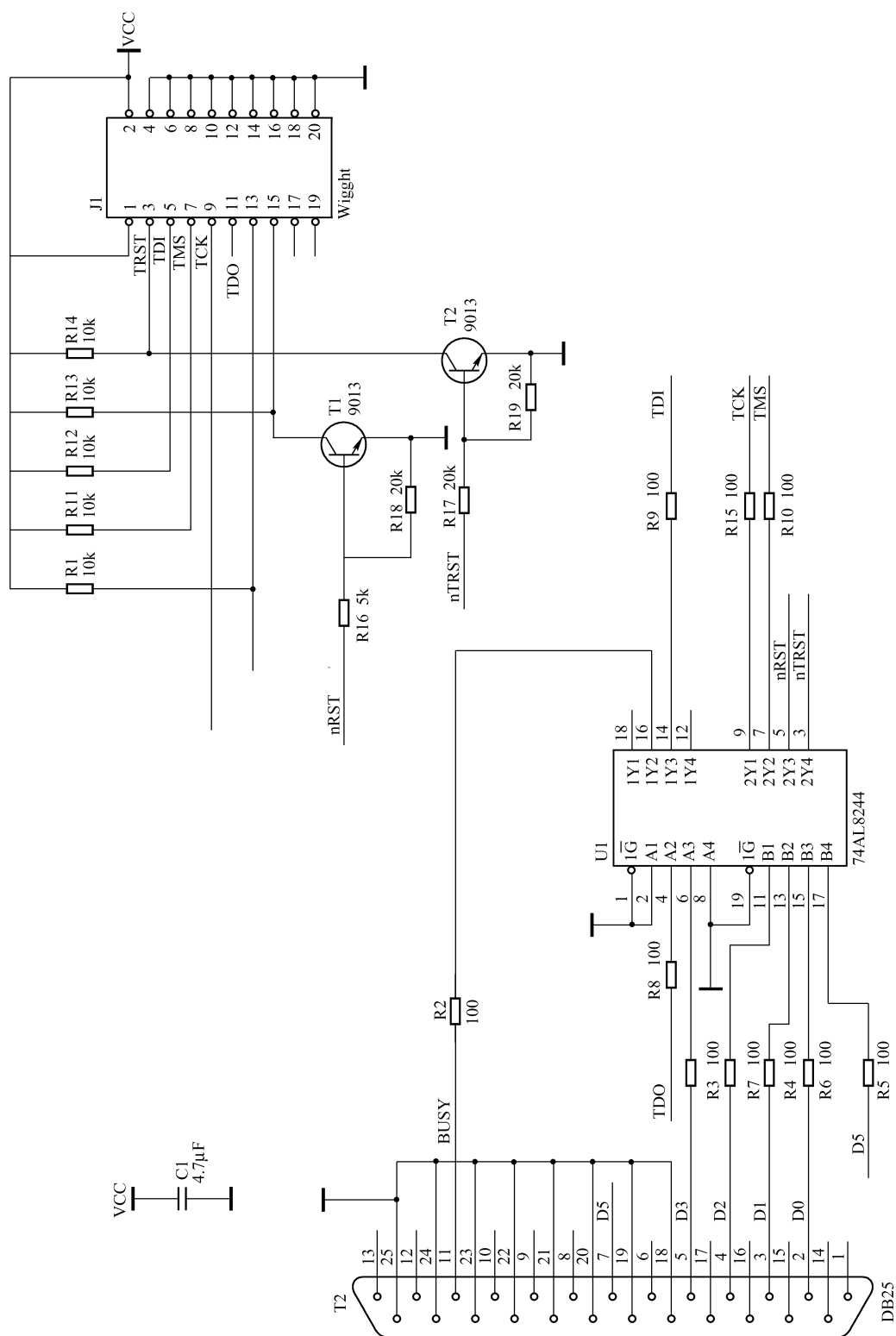


图 4-5 JTAG 的原理图

下面具体介绍 JTAG 原理图中的第一部分。

首先要打交道的是计算机的并口。简易仿真器的并口和计算机并口通过并口延长线连接。其次是 JTAG 接口，20 引脚的 JTAG 信号排列如图 4-6 所示，它必须和目标板上的 JTAG 接口对应。

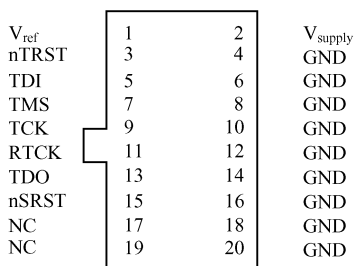


图 4-6 JTAG 接口的信号排列示意

接口是一个 20 引脚的 IDC 插座，表 4-2 给出了具体的信号说明。

表 4-2 JTAG 引脚说明

序 号	信 号 名	方 向	说 明
1	V_{ref}	Input	接口电平参考电压，通常可直接接 5V 电源
2	V_{supply}	Input	电源 5v
3	nTRST	Output	JTAG 复位（可选项）。在目标端应加适当的上拉电阻，以防止误触发
4	GND	—	接地
5	TDI	Output	Test Data In from Dragon-ICE to target
6	GND	—	接地
7	TMS	Output	Test Mode Select
8	GND	—	接地
9	TCK	Output	Test Clock output from Dragon-ICE to the target
10	GND	—	接地
11	RTCK	Input	Return Test Clock（可选项）。由目标端反馈的时钟信号，用来同步 TCK 信号的产生。不使用时可以直接接地
12	GND	—	接地
13	TDO	Input	Test Data Out from target to Dragon-ICE
14	GND	—	接地
15	nSRST	Input/Output	System Reset（可选项），与目标板上的系统复位信号相连。可以直接对目标系统复位，同时可以检测目标系统的复位情况。为了防止误触发，应在目标端加上适当的上拉电阻
16	GND	—	接地
17	NC		保留

续表

序 号	信 号 名	方 向	说 明
18	GND	—	接地
19	NC	—	保留
20	GND	—	接地

下面介绍目标系统如何设计。

目标板使用相同的 20 引脚针座，信号排列见表 4-2。RTCK 和 nTRST 这两个信号根据目标 ASIC 是否提供对应的引脚来选用，nSRST 则根据目标系统的设计考虑选择使用。图 4-7 是一个典型的连接关系图。

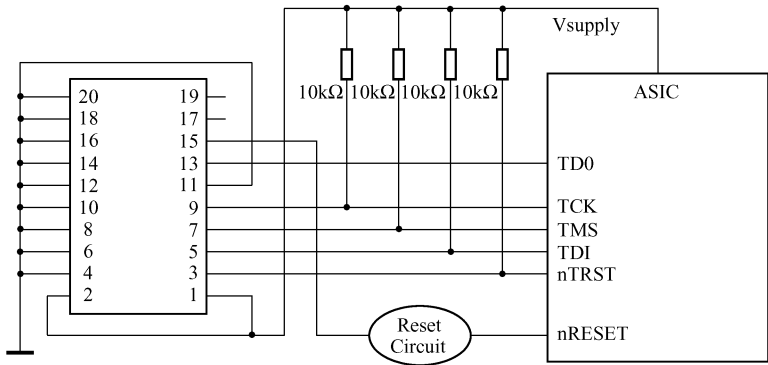


图 4-7 JTAG 和 CPU 的连接关系图

在目标系统的 PCB 设计中，最好把 JTAG 接口放置得离目标 ASIC 近一些，因为如果这两者之间的连线过长，会影响 JTAG 口的通信速率。

也有 14 引脚的 JTAG，那么 14 引脚的 JTAG 如何与 20 引脚的 JTAG 连接呢？当前一般使用工业标准的 20 引脚 JTAG 插头，但是有些老的系统采用一种 14 引脚的插座。这两类接口的信号排列如图 4-8 所示，它们之间的信号电气特性都是一样的，因此可以把对应的信号直接连起来进行转接。

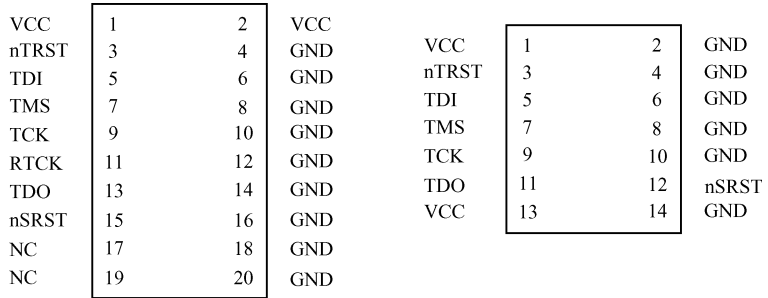


图 4-8 JTAG 转接

JTAG 的注意事项如下：

- (1) JTAG 仿真头和目标板之间的连接必须可靠，这是因为此类错误软件没有办法进行

检测和纠正。

(2) 由于信号的传输延迟, 仿真头和目标板之间的电缆长度是有限制的。因为在正常情况下, 仿真头在接收从目标板传回的数据时, 其采样时钟用的是 TCK, 它的工作频率比较高。如果传输延迟过长, 采样的数据将不正确。用自适应时钟可以解决此问题, 在这种工作模式下, 仿真头只有在接收到目标板的 RTCK 信号时才会采样 (从 TDO) 或者发送数据 (TDI)。

(3) 引脚的电流消耗问题。在 ARM CPU 上, JTAG 的信号引脚消耗的电流一般为 4mA, 这么小的驱动电流对于很短的 JTAG 引线已经够了。但如果引线加长, 则必须考虑传输线效应, 应该提供合适的阻抗匹配, 否则信号会畸变, 无法正常工作。

(4) 对于典型的目标板 JTAG 接口, 这是弱驱动电流, 不带阻抗匹配, 因此电缆长度不宜超过 20cm, 否则就要修改目标板的 JTAG 接口电路。采用以下方法可以将 JTAG 电缆延长到数米: 在 TDO、RTCK 上加 100Ω 的电阻, 并在 JTAG 口上加缓冲器, 当然具体长度还是和包含缓冲器在内的信号传输的延迟有关。如果 JTAG 接口电缆还要延长, 一种可行的方案是将 JTAG 信号通过差分驱动实现 (如 RS-422), 传输电缆改为多个双绞线, 考虑到传输延迟 RTCK 必须用上。理论分析表明, 采用这种方式的电缆长度可以达到几十米。

在了解了硬件部分以后, 下面就要了解软件部分是如何配合工作的。首先需要了解的是 ARM 的 ADP。

4.2.5 JTAG 仿真器驱动软件

前面提到的硬件必须配合软件才能工作, 软件需要完成的功能分为两大块:

(1) 在调试主机和调试目标之间建立通信连接。把调试主机发出的命令或数据送入 ARM 芯片内的 Embedbed-ICE, 把 Embedbed-ICE 的数据或者命令送到调试主机。在调试目标端, 数据的送入和送出通过 TDI 和 TDO 来实现。在主机端, 数据通过并口的两个引脚以串行的方式输入、输出。其实, 也可以采用串口或以以太网在调试主机和仿真头之间建立连接。不过这样就不能采用 Wiggler 电缆了, 仿真头的通信处理也变得复杂化。当然, 优点是数据传输快, 因为主机软件不必用并口来模拟串行数据的传输。

(2) 在调试主机和目标系统之间运行 ADP 协议。ADP 协议比较复杂, 以下详细介绍。

ADP 是 Angle Debug Protocol 的缩写, 在调试会话过程中, 它用来在调试主机和调试目标之间建立可靠的连接。协议首要的任务是让调试主机可靠和灵活地访问调试目标系统, 其数据链路层用点对点的协议实现。ADP 的基本工作方式是采用远程过程调用的 C/S 结构, 不过和其中的角色不是固定的。也就是在 host 和 debug target 之间可以建立多个数据通道, 在一个通道中 host 是 client, debug target 是 server, 在另一个通道中 debug target 是 client, host 是 server。通过数据链路层协议, ADP 协议可以支持的物理连接有多种, 包括串行接口、并行接口及网络接口。

ADP 协议结构: ADP 是 3 层架构, 从高到低依次为数据提供层 (DataProvider layer)、通道层 (Channel layer) 及设备层 (Device layer)。和标准的 ISO 网络协议比较, 数据提供层属于应用层协议, 通道层属于传输层协议, 设备层属于数据链路层协议。图 4-9 和图 4-10 是分别从调试主机和目标机角度看的协议层次, 可以发现该协议是对称的。

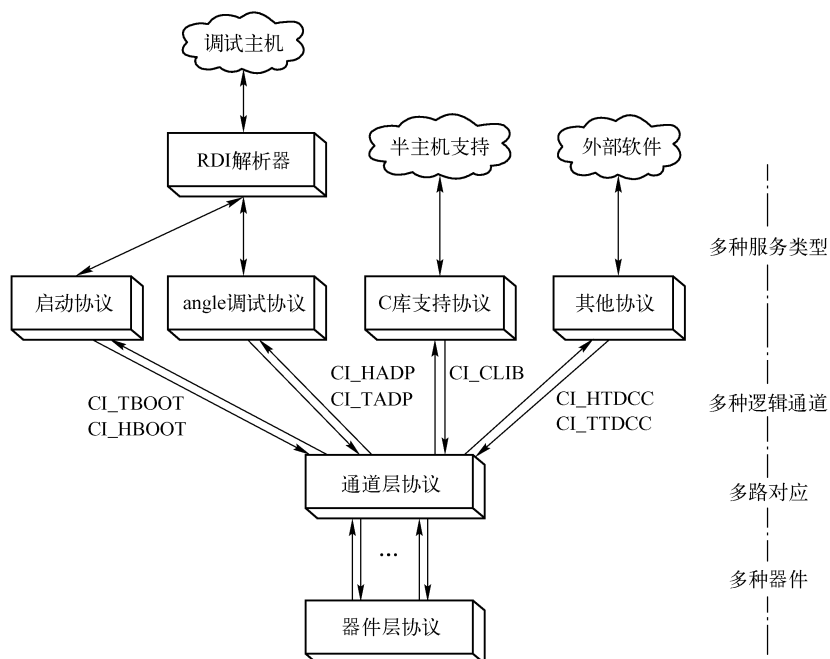


图 4-9 从调试主机角度看的协议层次

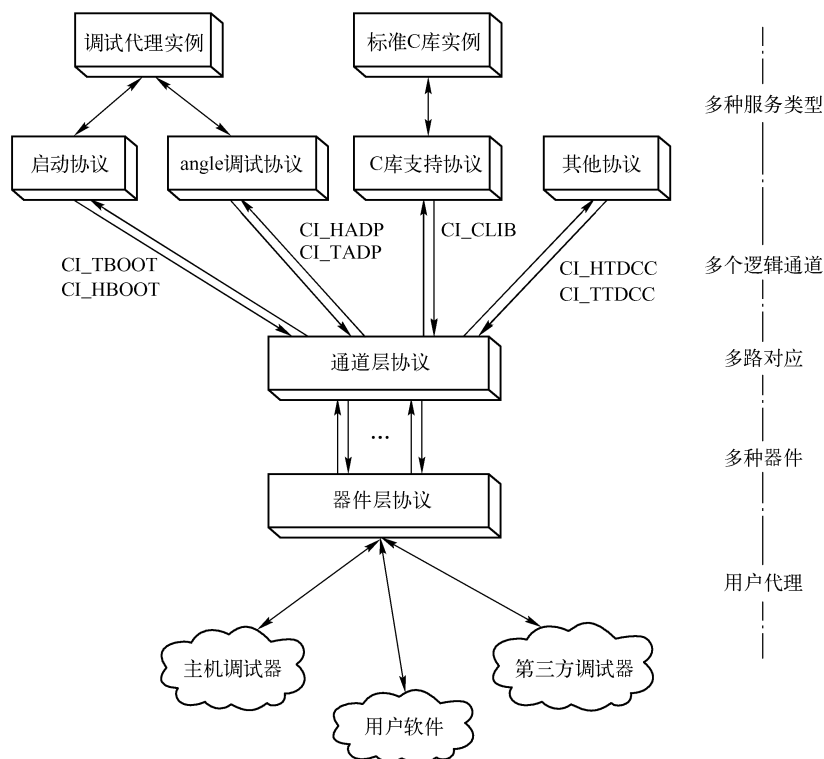


图 4-10 从调试目标机角度看的协议层次

以下对协议的每一层进行分析。

1. 数据提供层

ADP 数据提供层数据包的格式如图 4-11 所示。

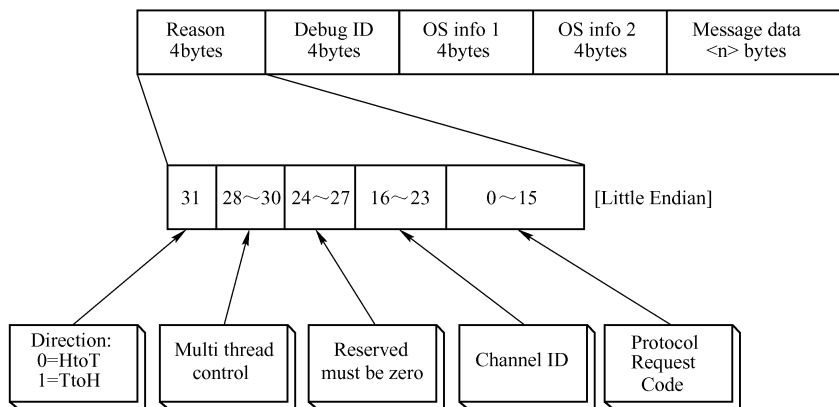


图 4-11 ADP 数据提供层数据包格式

图 4-11 中，第一个字段是 4 字节的 **Reason Code**，它定义了接收端要执行的操作，并且也可以用来检查应答类型。发送和应答消息的不同仅仅在于 **MSB** 的 **Direction bit** 不同。

在调试系统是多线程场合，**Debug ID** 是提供给主机的调试软件使用的，目标机根据主机的请求 **ID** 在其回应中包含相同 **ID**。如果调试系统仅仅是单线程，就不需要 **ID**，但 **ID** 必须固定为 **0xFFFFFFFF**。

OS info 1 和 **OS info 2** 用在多线程操作系统场合，用来确定当前函数调用属于哪个线程。如果仅仅在单线程场合，则必须设置为 **0xFFFFFFFF**。

流控：每个服务都有两个通道，即请求通道和应答通道。应答通道的应答信息和请求通道的信息一一对应。

可靠性及错误检测：高层的协议假设通道是不会发生错误的，错误由下层协议处理。

操作：操作码由相应的协议来定义（**ADP**、**BOOT**、**Clib**），不同协议的数据包通过不同的通道传输。**Reason Code** 的最高位用来表示是从主机向目标机发送还是从目标机向主机发送。当然，根据方向的不同，数据的格式有一定差异。

2. 通道层

通道层协议负责将数据包从通信介质一端经过一系列通道传递到另一端。其实，这里的通道和 **TCP/IP** 协议中的端口是等价的。该层上面的接口是基于数据包的，上层软件通过接口把数据包送给本层，并且认为本层将按顺序发送数据包。通道层在上层软件提交数据包的时候明确要求提供数据到达信息的情况下，才会向上层发送数据包到达。通道层下面的接口是通信介质，如串行电缆就能在不需要地址信息的情况下把数据从一端传输到另一端。通道层认为通信都是点对点的，认为数据包的内容在通信媒体中也是按次序提交的。

数据格式：通道层协议用来确定目前的数据包应该走哪一个通道，并且协议包含数据包的顺序号，如图 4-12 所示。目前的协议头部有 4 字节，分别如下。

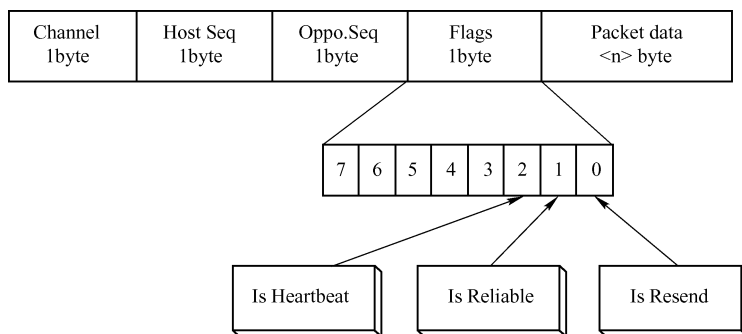


图 4-12 通道层协议

1) 通道 ID

通道 ID 在数据包刚刚接收到的时候被检查，它必须小于预先定义的一个极限。收发双方在开始工作以前应该对通道 ID 的子集和含义进行协商。

2) 主机数据包顺序号及目标数据包应答顺序号

主机数据包和应答数据包的顺序号用来确定发送和接收双方的相应状态（例如，如果发送方发送的数据包提前超过接收方当前接收的滑动窗口下限，接收方就需要请求重传）。数据包的发送和接收可以参考滑动窗口协议。

3) flag 字节

flag 字节用来确定不同的数据包类型。3 个 flag bit 是互斥的。一共有 4 种数据包类型：Resend、Heartbeat、Reliable data、Unreliable data（“Datagram”）。

Boot 代理：Boot 代理通过通道层可靠的通信在主机和目标机之间建立通信。包括确定数据传输的装置，同步主机和目标机及会话参数协商（如最大消息长度）。一次会话结束，Boot 代理必须将系统状态置为允许下一次会话开始。所有基于主机通信的 Angle 调试系统都应该提供 Boot 代理。Boot 代理需要处理以下几种情况：

- 目标板比主机先启动；
- 目标板比主机晚启动；
- 会话开始时收发双方发送包的流程；
- 会话结束时收发双方发送包的流程；
- 数据包序号的复位。

3. 设备层

设备层的通信协议和具体通信的设备相关。例如，采用以太网通信的场合协议采用的是 UDP；在设备驱动程序直接和硬件通信的场合则采用其他协议，这就是基于字节流的点对点通信协议。

设备层的通信协议如图 4-13 所示。

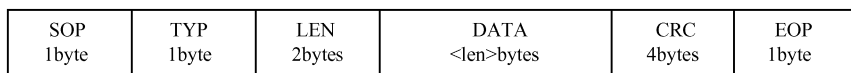


图 4-13 设备层通信协议

这一层的数据包用来对通道层递交的数据包进行封装、加入字段、错误检测及填充字符等操作。为了防止数据包在接收端处理时包中的净载荷字节和这些控制信息段的字节混淆，应对净载荷的字节采用字符填充。这些字符包含如下几个：

- Start of Packet 0x1C;
- End of Packet 0x1D;
- Escape 0x1B;
- Stop Sending (XOFF) 0x13;
- Start Sending (XON) 0x11。

可靠性和错误检测：协议检测一系列的错误并且向上层提交。具体包括：

- 帧错误；
- 长度错误；
- CRC 错误。

当检测到帧错误以后，接收方请求重传。

4.3 ADS 开发套件

目前，市场上的 ARM 集成开发环境有多种，如 ARM 公司自身推出的 SDT、ADS 及 RealView，IAR Systems 公司的 IAR 开发套件，Micetec 的 Hitool、JEDI View 等。这些开发软件的基本功能都差不多，但各有优势。有些专长于调试内核级程序，有些专长于调试任务级程序。任务级和内核级调试的不同在于：任务级调试中断的是某个线程，操作系统不会停止；而内核级调试中断所有程序。本节介绍比较常用的、国内使用者最多的 ADS 1.2 开发套件。由于 ADS 1.2 是由 ARM 公司开发的，因此优化的程度与效率都比较高。

ADS 包含两大部分：Codewarrior 和 AXD。Codewarrior 提供工程文件的创建、管理、编辑、编译及编译配置等，AXD 则提供调试。通过本节的学习，读者将掌握如何在 Codewarrior 中创建一个工程，并且用自制的简易 JTAG 仿真头来调试软硬件系统。

4.3.1 在 ADS 1.2 中使用简易 JTAG 仿真头调试

首先按照图 4-14 所示连接仿真头、计算机及实验板，然后安装软件，软件在本书光盘里有提供。

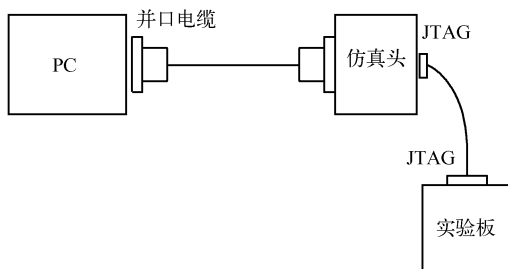


图 4-14 JTAG 仿真头和 PC 连接

执行以下步骤:

- (1) 安装调试代理软件;
- (2) 安装 ADS 1.2;
- (3) 进行环境配置;
- (4) 打开源程序, 编译并且运行。

简易仿真头采用前面介绍的 Wiggler 电缆, 调试代理软件的安装及和 ADS 的连接如下:

(1) 首次使用时先安装驱动(以后不用再安装)——执行文件下的安装驱动.exe, 安装好并口驱动和 OCX;

(2) 安装完毕后即可运行 ARM 7 Agent 或 ARM 9 Agent, 调试 ARM 7 系统或 ARM 9 系统;

(3) 在 ADW(SDT)/AXD(ADS)的调试配置选项里选择 remote_a.dll;

(4) 在 ADW/AXD 的调试配置里, IP 地址必须填写 127.0.0.1。

首先, 安装代理软件。

在光盘中找到调试代理目录, 安装并口驱动, 出现如图 4-15 所示界面。

单击 “Install” 按钮, 出现如图 4-16 所示界面, 说明驱动安装成功。



图 4-15 安装调试代理并口驱动



图 4-16 调试代理并口驱动安装成功

打开调试代理目录下的 arm7.exe。如果 JTAG 接口没有连接好, 或者 ARM 开发板没有上电, 就会出现如图 4-17 所示界面。

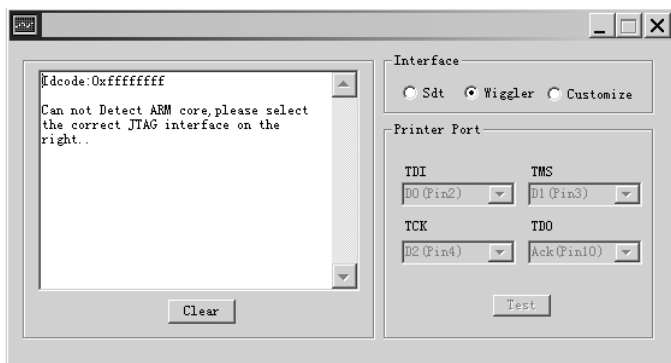


图 4-17 JTAG 的一种故障提示

如果成功则出现成功检测到 ARM 内核的界面。

然后，开始配置 ADS 开发环境。

打开 AXD 程序，选择菜单命令“Options”→“Configure Target”，出现如图 4-18 所示对话框。

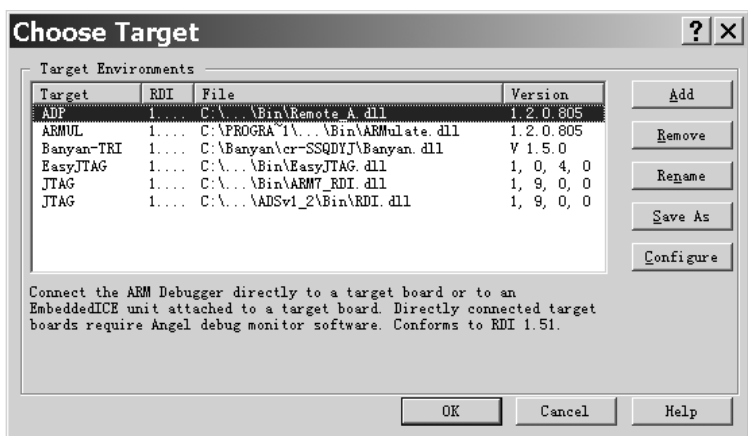


图 4-18 ADP 的设置 (1)

在“Choose Target”对话框中选择“ADP”，然后单击“Configure”按钮，出现如图 4-19 所示对话框。

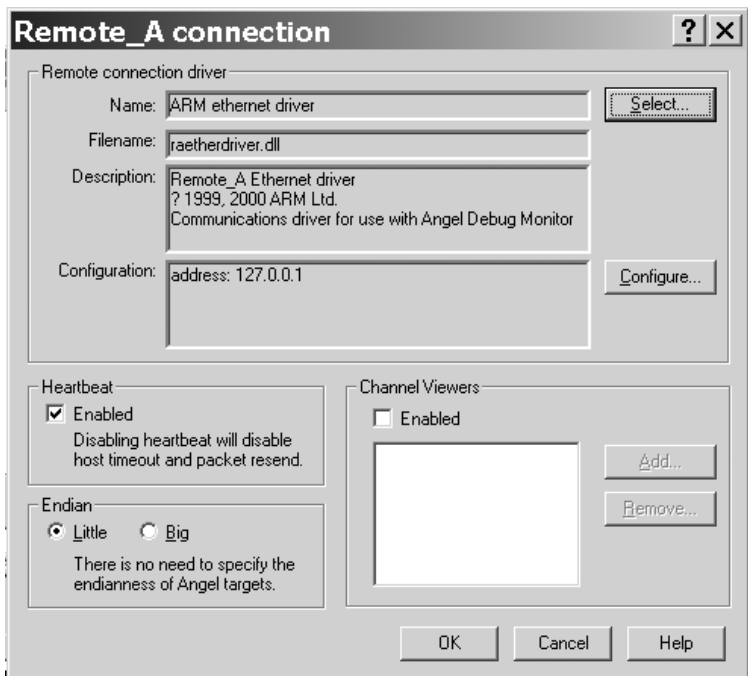


图 4-19 ADP 的设置 (2)

单击“Select”按钮，出现如图 4-20 所示对话框。

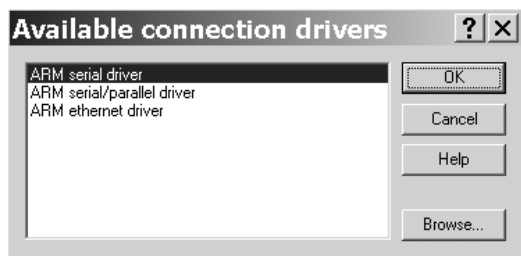


图 4-20 ADP 的设置 (3)

选择“ARM ethernet driver”，单击“OK”按钮确认，返回如图 4-21 所示界面。

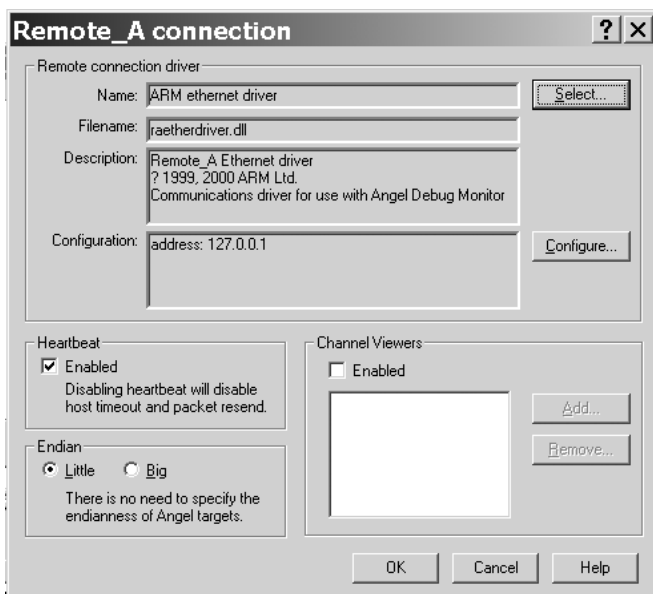


图 4-21 ADP 的设置 (4)

单击“Configure”按钮，出现如图 4-22 所示对话框。



图 4-22 ADP 的设置 (5)

在“Target IP address”文本框输入“127.0.0.1”，然后单击“OK”按钮就可以了。这时如果没有打开调试代理软件，AXD 会出现失败。因此必须先打开调试代理软件，然后再运行 AXD 才能正常。

4.3.2 ADS 中程序的调试

打开一个现有的源程序，并且运行。

启动 CodeWarrior for ARM Developer Suite，如图 4-23 所示。

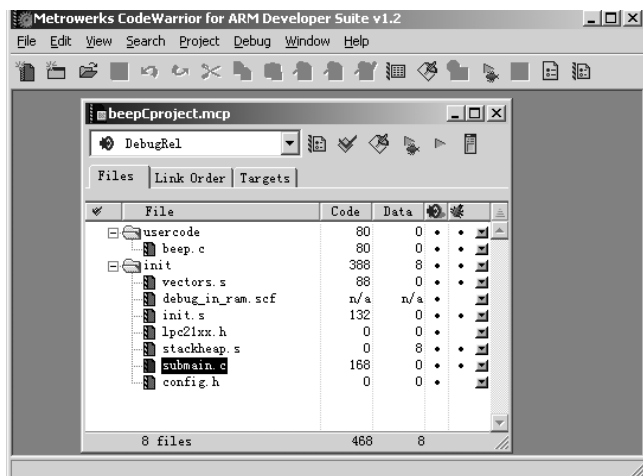


图 4-23 CodeWarrior 中打开工程

在 File 下选择一个现有文件。对本例而言，将实验程序全部复制到 D 盘，选择“D:\实验”目录下的 beep 文件夹，打开 beepCproject.mcp，工程将出现在主界面下。在 beepCproject 下有一些调试选项：Debug、DebugRel、Release。它们分别对应应在 RAM 中运行程序，在 Flash 中运行程序，其不同仅在于 Debug Setting 的配置上。这里选择 Debug，也就是在 RAM 中运行程序，此时源程序自动编译。查看调试代理目前是否正常运行，在 Project 中选择 Debug，这时 AXD 将自动运行，并停在当前指令入口处，也就是复位后的地址处，如图 4-24 所示。

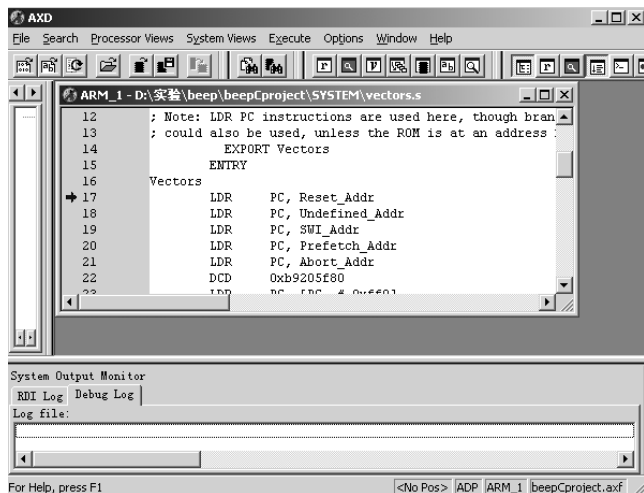


图 4-24 AXD 下面的调试

对于 LPC 2132 而言, 停留在 0x40000000 处, 这里放的是“LDR PC, Reset_Addr”, 也就是管理模式下的第一条指令。

在图 4-24 所示界面中右击, 出现快捷菜单, 如图 4-25 所示, 选择“Interleave Disassembly”, 就能显示当前对应的汇编代码、机器码及指令地址。

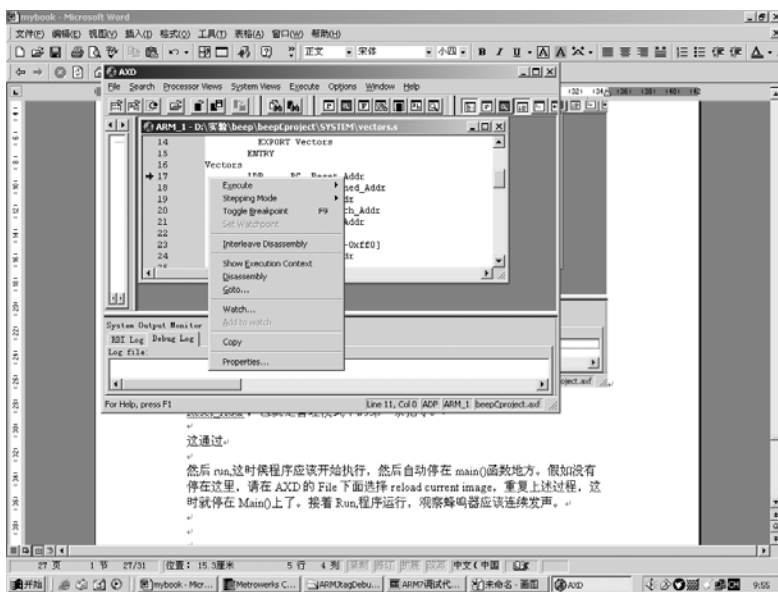


图 4-25 AXD 汇编查看

图 4-26 所示就是当前打开的源程序对应的汇编语言。可以看到, 第二条指令地址为 40000004, 第三条指令地址为 40000008, 由于每个异常矢量为 4 字节, 因此可以计算出第一条指令地址是 40000000。

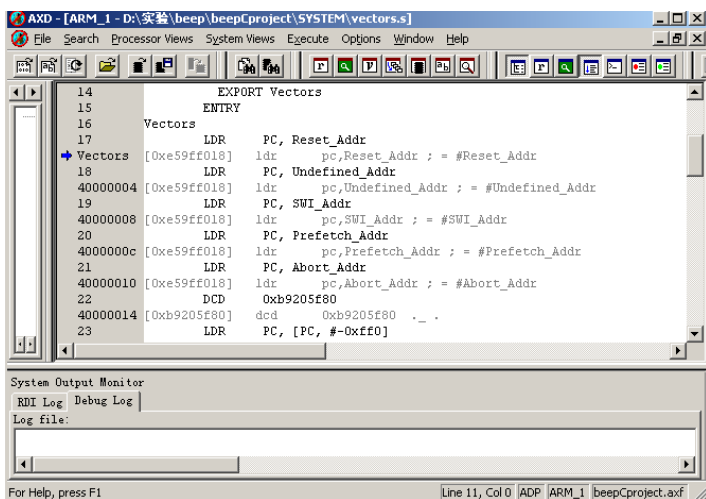


图 4-26 对应 C 语言的汇编入口

然后运行，这时程序应该开始执行，之后自动停在 `main()` 函数处。假如没有停在这里，请在 AXD 的 File 下面选择 Reload Current Image，重复上述过程，就停在 `main()` 处了。接着运行程序，观察蜂鸣器，应该连续发声。

AXD 的使用涉及一些基本项，如 Step, Step in, Step out, Go、Run to cursor 等，这里就不再提了，此处要注意的是如何查看程序中的一些变量和状态。变量的查看在 Processor Views 下可以找到，常用的有 Register, Variables 及 Watch。Register 可以查看和修改各种异常状态的 ARM 寄存器，如果要把程序复位到 PC 起始处，可直接修改当前的 PC 值（例如：PC=0x40000000），通过查看 CPSR 可以获知目前系统是否允许进入 IRQ 和 FIQ。请不要随便修改 SPSR，因为它是用来保存进入异常前的处理器状态的。在 Variables 里包含了局部变量和全局变量查看与修改功能。如果要看局部变量的数值，则只有当程序进入局部变量所在的函数时才能看到；而全局变量在任何时候都可以看到。Watch 可以通过表达式方式来查看某个变量或者寄存器的数值，这样就更加方便了用户的特殊需求。Memory 查看内存空间的数据，可以修改对应位置的数值，当然千万不要修改程序代码，否则将不能正常调试。Low Level Symbols 可以查看在程序汇编语言里定义的标识符，这也包括 C 语言库初始化函数 main 中的一些定义。Console 用来显示 Semihosting 方式的输出。由于 Armulate、Multiice 及 Angle 调试器支持这个功能，用户可以直接使用，不用写其他代码。Semihost 功能只在调试时使用，独立运行的目标板不能包括 Semihost 函数。因此需要做一些移植工作，把 Semihost 涉及的与目标相关的底层函数替换掉。提供 Semihost 是为了调试时方便，如果觉得不容易掌握，可以完全不用它，直接编写硬件驱动及对应的接口。比较现实的一种方法是在硬件没有做出来的时候用 Armulate 调试，这时一般就要用到 Semihost；因为涉及输入/输出这些基本功能（如 `printf()`）；当硬件完成以后，用前面提到的 JTAG 仿真器调试，此时修改底层驱动代码。例如，`printf()` 通过调用 `fputc()`、`fputc()` 调用 `WriteC()` 向控制台输出，只要修改 `fputc()` 的调用，让 `fputc()` 调用 `sendchar()` 完成实际的输出即可，而 `Sendchar` 是用户自己编写的串口输出驱动函数。

4.4 ARM 启动代码和 Bootloader

通常将处理器复位开始执行第一条指令到进入 C 语言的 Main 函数之前执行的那段汇编代码称为启动代码。而 Bootloader 不是一段代码，它是一个具有引导装载功能的完整的程序项目，如可以引导装载 Linux 或 Wince 的 vivi、uboot 等，它包含了启动代码部分。

如果程序全部用汇编语言，就不需要启动代码，只需对系统进行初始化，这和用单片机进行汇编开发类似。但如果需要用 C 语言编写程序，则必须提供启动代码。也许有些人会觉得奇怪，为什么用 Keil C51 不必写任何启动代码，但在 ARM 下编程却要写？这是因为 ARM 公司自己不生产 CPU，只出售芯片技术，许多公司扩充 ARM 内核，形成了具有各自特色的 ARM 处理器。例如，ROM、RAM 的地址安排各不一样，片内外设也各不相同。而 C51 的启动代码已经在 Keil C51 里集成，用户不需要自己另外再写。你可能会问：为什么一定要用汇编写启动代码而不直接用 C 语言来初始化呢？因为 C 语言在工作以前必须初始

化一些 C 语言的库,例如,为 C 程序分配栈空间。当这些基本项目没有初始化时,C 语言的程序是无法工作的。虽然 C 语言最终被编译器编译为汇编代码,但只有基本的设置完成,这些代码才能正常运行,而 ARM 的编译器不会做这些初始化。

编写启动代码需要有如下的预备知识:

- 了解 ARM 异常的种类;
- 了解各种异常对应的堆栈分配方法;
- 了解 CPSR 和 SPSR;
- 了解汇编程序与 C 程序的调用;
- 了解 ADS 编译器相关的一些知识;
- 了解某些系统相关的外设设置。

在 ARM 系统上电复位后,需要设置中断向量表、初始化各模式堆栈、设置系统时钟频率等,而这些过程都是针对 ARM 内部寄存器结构的操作,用 C 语言编程是很难实现的。因此在转到应用程序的 C/C++编写之前,需要用 ARM 的汇编语言编写启动代码,由启动代码完成系统初始化及跳转到用户 C 程序。在 ARM 的设计开发中,启动代码的编写是一个极重要的过程。启动代码随具体的目标系统和开发系统有所区别,但通常包含以下部分:

- 向量表定义;
- 地址重映射及中断向量表的转移;
- 堆栈初始化;
- 设置系统时钟频率;
- 中断寄存器的初始化;
- 进入 C 应用程序。

4.4.1 启动代码主要构成

下面结合 PHILIPS 的 LPC 2119 的启动代码来分析与说明 ARM 7 处理器的启动代码的编写。

1. 向量表定义

ARM 芯片上电或复位后,系统进入管理模式、ARM 状态,PC (R15) 指向 0x00000000 地址处。中断向量表为每一个中断设置 1 个字的存储空间,存放一条跳转指令,通过这条指令使 PC 指针指向相应的中断服务程序入口,继而执行相应的中断处理程序。LPC 2219 的中断向量表和其他基于 ARM 核的芯片中断向量表较类似,只是要注意,LPC 2219 要使向量表所有数据 32 位累加和为零 (0x00000000~0x0000001C 的 8 个字的机器码累加),才能使用户的程序脱机运行。

2. 地址重映射及中断向量表的转移

ARM 7 处理器在复位后从地址 0 读取第一条指令并执行,因此系统上电后地址 0 必须是非易失的 ROM/Flash,这样才能保证处理器有正确可用的指令。为了加快对中断的处理及实现在不同操作系统模式下对中断的处理,需要重新映射中断向量表、Bootblock 和 SRAM 空间的一小部分。ARM 具有非常灵活的存储器地址分配特性。ARM 处理器的地址重映射

机制有以下两种情况。

(1) 由专门的寄存器完成重映射 (Remap)，只需对相应的 Remap 寄存器相应位进行设置即可。

(2) 没有专门的 Remap 控制寄存器，需要重新改写用于控制存储器起始地址的块 (Bank) 寄存器来实现 Remap。在 LPC 2119 上的重映射，可以通过存储器映射控制器来实现。实现 Remap 操作的程序如下：

```
MOV R8, #0x40000000;      /设置新向量表起始地址/
LDR R9, =Interrupt_Vector_Table; /读原向量表源地址/
LDMIA R9!, (R0-R7); /复制中断向量表及中断处理程序的入口地址到 RAM 中 (64 字节) /
STMIA R8!, (R0-R7)
LDMIA R9!, (R0-R7)
STMIA R8!, (R0-R7)
LDR R8, =MEMMAP; /Remmap 操作/
MOV R9, #0x02
STR R9, [R8]
```

3. 堆栈初始化

启动代码中各模式堆栈空间的设置是为中断处理和程序跳转时服务的。当系统响应中断或程序跳转时，需要将当前处理器的状态和部分重要参数保存在一段存储空间中，所以对每个模式都要进行堆栈初始化工作，给每个模式的 SP 定义一个堆栈基地址和堆栈的容量。堆栈的初始化有两种方法：第一种方法是结合 ADS 开发套件中的分散加载文件来定义堆栈。第二种方法是最简单也是最常用的，就是直接进入对应的处理器模式，为 SP 寄存器指定相应的值。下面给出用第二种方法初始化管理模式和中断模式堆栈的程序：

```
MSR CPSR_c, #0xD3; /切换到管理模式，并初始化管理模式堆栈/
LDR SP, Stack_Svc
MSR CPSR_c, #0xD2; /切换到 IRQ 模式，并初始化 IRQ 模式的堆栈/
LDR SP, Stack_Irq
...
```

4. 系统部分时钟初始化

时钟是芯片各部分正常工作的基础，应该在进入 main() 函数前设置。部分 ARM 7 片子内部集成有 PLL (锁相环) 电路，用户可以用低频率的晶振通过 PLL 电路获得一个较高频率的时钟。LPC 2119 内部的 PLL 电路接收的输入时钟频率范围为 10~25MHz，输入频率通过一个电流控制振荡器 (CCO) 倍增到 10~60MHz。同时为了使高速的 ARM 处理器与低速的外设正常通信和降低功耗 (降低外设运行速度使功耗降低)，LPC 2119 又集成了一个额外的分频器。PLL 的激活是由 PLLCON 寄存器控制的。PLL 倍频器和分频器的值由 PLLCFG 寄存器控制。对 PLLCON 或 PLLCFG 寄存器的更改必须遵循严格的顺序，否则所作更改是无法生效的 (在连续的 VPB 周期内向 PLLFEED 寄存器写入 0xAA、0x55，在此期间中断必须是被禁止的。)

5. 中断初始化

ARM 7 的向量中断控制器 (Vectored Interrupt Controller) 可以将中断编程为 3 类: FIQ、向量 IRQ、非向量 IRQ。FIQ 中断请求的优先级最高, 其次是向量 IRQ 中断请求, 非向量 IRQ 中断请求的优先级最低。VIC 具有 32 个中断请求输入, 但在 LPC 2219 中只占用了 17 个中断输入。对于这 17 个中断源的 IRQ/FIQ 选择, 由 VICIntSelect 寄存器控制, 当对应位设置为 1 时, 此中断为 FIQ 中断, 否则为 IRQ 中断。若再将 IRQ 中断设置到向量控制寄存器 (VICVectCntIn) 中, 则此中断为向量 IRQ 中断, 否则为非向量 IRQ 中断。FIQ 中断是专门用来处理那些需要及时响应的特殊事件的, 应尽可能地只给 FIQ 分配一个中断源。

6. 进入 C 应用程序

至此, 系统各部分的初始化基本完成, 可以直接从启动代码转入应用程序的 main() 函数入口。从启动代码转入应用程序的实例代码如下:

```
IMPORT main
LDR R0, =main
BX R0
```

启动代码的编写可谓千变万化, 即使同一型号的 CPU 启动代码也不一样, 这是由编写启动代码的风格和系统的要求决定的。但总的来说, 启动代码的工作流程都是一样的, 掌握了编写的要领, 也就能自如地写出适合自己需要的启动代码。总结以上代码, 可以得出启动代码编写的步骤如下:

- (1) 初始化异常向量;
- (2) 初始化 CPU 芯片;
- (3) 初始化目标板;
- (4) 初始化堆栈;
- (5) 调用 _main() 函数初始化 C 语言库。

另外还有一种风格的启动代码, 就是 PHILIPS 的 LPC 2xxx 系列启动代码。

4.4.2 启动代码实例分析

```
#处理器的 7 种工作方式的常量定义
.EQU    Mode_USR,   0x10      #用户模式
.EQU    Mode_FIQ,   0x11      #FIQ 模式
.EQU    Mode_IRQ,   0x12      #IRQ 模式
.EQU    Mode_SVC,   0x13      #超级用户模式
.EQU    Mode_ABT,   0x17      #终止模式
.EQU    Mode_UND,   0x1B      #未定义模式
.EQU    Mode_SYS,   0x1F      #系统模式

#中断屏蔽位
.EQU    I_Bit,      0x80      //IRQ 中断控制位, 当被置位时, IRQ 中断被禁止
.EQU    F_Bit,      0x40      //FIQ 中断控制位, 当被置位时, FIQ 中断被禁止
```

```

#状态屏蔽位
.EQU    T_bit,      0x20      //T 位，置位时在 Thumb 模式下运行，清零时在 ARM 下运行

#定义程序入口点
.globl _start

        .code 32
        .TEXT

_start:

#中断向量表
Vectors:

        LDR    PC, Reset_Addr  //把 Reset_Addr 地址的内容放入 PC 中
        LDR    PC, Undef_Addr
        LDR    PC, SWI_Addr
        LDR    PC, PAbt_Addr
        LDR    PC, DAbt_Addr
        .long  0xb9205f80      @ keep interrupt vectors sum is 0
        LDR    PC, [PC, #-0xff0] //当前 PC 值减去 0xFF0 等于 IRQ 中断入口地址
        LDR    PC, FIQ_Addr

#地址表
Reset_Addr:                                #该地址标号存放 Reset_Handler 程序段的入口地址
        .long  Reset_Handler
Undef_Addr:                                #该地址标号存放 Undef_Handler 程序段的入口地址
        .long  Undef_Handler
SWI_Addr:                                  #该地址标号存放 SWI_Handler 程序段的入口地址
        .long  SWI_Handler
PAbt_Addr:                                 #该地址标号存放 PAbt_Handler 程序段的入口地址
        .long  PAbt_Handler
DAbt_Addr:                                 #该地址标号存放 DAbt_Handler 程序段的入口地址
        .long  DAbt_Handler
        .long  0
IRQ_Addr:                                  #地址标号处存放一个无效的数据
        .long  0
FIQ_Addr:                                  #该地址标号存放 FIQ_Handler 程序段的入口地址
        .long  FIQ_Handler

Undef_Handler:
        B      Undef_Handler
PAbt_Handler:
        B      PAbt_Handler
DAbt_Handler:
        B      DAbt_Handler

```

#软中断的中断服务子程序入口地址

SWI_Handler:

```

STMFD    sp!, {r0-r3, r12, lr}    //入栈, 现场数据保护
MOV      r1, sp                    //把堆栈指针 SP 存入 R1 中
MRS      r0, spsr                  //把 SPSR 值存入 R0 中, SPSR 值为产生软中断
                                   //时的 CPSR
TST      r0, #T_bit                //判断 R0 (SPSR) 的 T 位是否为 0
#SPSR 的 T 位不为 0, 工作在 Thumb 模式下
LDRNEH   r0, [lr, #-2]             //SPSR 的 T 位不为 0, 则[lr-2]-> r0
BICNE    r0, r0, #0xFF00           //SPSR 的 T 位不为 0, 清除 r0 的 Bit8~Bit15 位
#SPSR 的 T 位为 0, 工作在 ARM 模式下
LDREQ    r0, [lr, #-4]             //SPSR 的 T 位为 0, 则[lr-4]-> r0
BICEQ    r0, r0, #0xFF000000       //SPSR 的 T 位为 0, 清除 r0 的 Bit24~Bit131 位

# R0 is interrupt number           //R0 是中断号
# R1 is stack point                //R1 是堆栈指针

BL        SWI_Exception            //进入软中断处理程序
LDMFD    sp!, {r0-r3, r12, pc}^    //出栈, 现场数据恢复

```

#快速响应中断的中断服务程序的入口地址

FIQ_Handler:

```

STMFD    SP!, {R0-R3, LR}          //入栈的现场保护
BL        FIQ_Exception            //进入 FIQ 的中断处理程序
LDMFD    SP!, {R0-R3, LR}          //出栈, 恢复现场
SUBS     PC, LR, #4                //返回主程序

```

#复位后程序处理的入口地址

Reset_Handler:

```

BL        RemapSRAM                //进行存储器映射的操作

```

#下面几行代码用来判断当前的工作模式

```

MRS      R0, CPSR                  //读 CPSR 到寄存器 R0
AND      R0, R0, #0x1F             //R0 = R0 AND 0x1F
CMP      R0, #Mode_USR             //比较 R0 和 #Mode_USR, 二者相减

```

//如果相等则说明当前处在用户模式下, 需要通过产生 11 号软中断进入系统模式。因为下面的初始化堆栈

//需要在不同的工作模式下切换, 而在用户模式下不能直接切换, 只有系统模式可以, 所以要通过产生 11 号软中断切换到用户模式

//

```

SWIEQ    #11
BL        InitStack                //进行堆栈初始化工作

```

#-----

#- 初始化 C 变量

#-----

```

#- 下表由连接器自动产生
#- RO: 只读=代码区
#- RW: 可读可写=预先初始化的数据（初始化的全局变量）和预先被清零的数据（未初始化的全局变量）
#- ZI: 预先被清零的数据区（未初始化的全局变量）
#- 预先被初始化的数据区定位在代码区之后
#- 预先被清零的数据区定位在预先被初始化的数据区之后
#- 注意数据区的位置
#- I 如果用 ARM SDT, 当链接器选择 no -rw-base 时, 数据区被映射在代码区之后
#- 可以把数据区放在内部的 SRAM( -rw-base=0x40 or 0x34)中
#- 或者放在外部的 SRAM( -rw-base=0x2000000 )中
#- 为了提高代码的密度, 预先被初始化的数据必须尽可能地少
#-----
#该部分程序功能: 先判断当前是在 RAM 中运行还是在 Flash 中运行, 如果在 Flash 中运行, 则
先把 flash 中的预先赋值的 RW 段数据和未赋值的 ZI 段数据都搬移到 RAM 区中, 再把 ZI 段数
据全部清零; 如果程序就是在 RAM 中运行, 则直接把 ZI 段数据清零

        .extern      Image_RO_Limit      /*ROM 区中数据段的起始地址*/
        .extern      Image_RW_Base       /*RW 段的起始地址*/
        .extern      Image_ZI_Base       /*ZI 段的起始地址*/
        .extern      Image_ZI_Limit      /*ZI 段的结束地址加 1 */

        ldr          r0, =Image_RO_Limit /*取 ROM 区中数据段的首地址*/
        ldr          r1, =Image_RW_Base  /*取 RAM 区中 RW 段的目标首地址*/
        ldr          r3, =Image_ZI_Base  /*取 RAM 区中 ZI 段的首地址*/
        cmp          r0, r1              /*比较 ROM 区中数据段首地址和 RAM
区中 RW 段目标首地址*/

        beq          NoRW                /*相等代表当前是在 RAM 中运行*/
        LoopRW:    cmp          r1, r3    /*不相等则和 RAM 区中 ZI 段的目标地址
比较*/

                                /*如果 r1<r3, 执行以下 3 条语句*/
        ldrc         r2, [r0], #4        /*把 r0 地址中的数据读出 r2 中, 然后 r0=r0+4*/
        strcc        r2, [r1], #4        /*把 r2 内的数据写入 r1 地址中, 然后 r1=r1+4*/
        bcc          LoopRw              /*跳转到 LoopRW 继续执行*/
        NoRW:        ldr          r1, =Image_ZI_Limit /*取 ZI 段的结束地址*/
        mov          r2, #0              /*将 r2 赋 0*/
        LoopZI:      cmp          r3, r1    /*将 ZI 段清零*/
        strcc        r2, [r3], #4        /*如果 r3<r1, 则将 r2 的内容写入 r3 地址单元
中, 然后 r3=r3+1*/

        bcc          LoopZI              /*如果 r3<r1（即 C=0）, 则跳转到 LoopZI */

        .extern      Main                /*声明外部变量*/
        B            Main                /*跳转到用户的主程序入口*/

#为每一种模式建立堆栈, ARM 堆栈指针向下生长
InitStack:

```

```

MOV    R1, LR           //把孩子程序返回地址保留在 R1 中

LDR     R0, =Top_Stack   //取栈顶地址到 R0 中
#进入未定义模式，并禁止 FIQ 中断和 IRQ 中断
MSR     CPSR_c, #Mode_UND|I_Bit|F_Bit
#设置未定义模式下堆栈的栈顶指针
MOV     SP, R0
SUB     R0, R0, #UND_Stack_Size      #未定义模式下堆栈深度

#进入终止模式，并禁止 FIQ 中断和 IRQ 中断
MSR     CPSR_c, #Mode_ABT|I_Bit|F_Bit
#紧接着未定义模式下的堆栈，设置终止模式下的栈顶指针
MOV     SP, R0
SUB     R0, R0, #ABT_Stack_Size      #终止模式下堆栈深度

#进入 FIQ 模式，并禁止 FIQ 中断和 IRQ 中断
MSR     CPSR_c, #Mode_FIQ|I_Bit|F_Bit
#紧接着终止模式下的堆栈，设置 FIQ 模式下的栈顶指针
MOV     SP, R0
SUB     R0, R0, #FIQ_Stack_Size      #FIQ 模式下的堆栈深度

#进入 IRQ 模式，并禁止 FIQ 中断和 IRQ 中断
MSR     CPSR_c, #Mode_IRQ|I_Bit|F_Bit
#紧接着 FIQ 模式下的堆栈，设置 IRQ 模式下的栈顶指针
MOV     SP, R0
SUB     R0, R0, #IRQ_Stack_Size      #IRQ 模式下的堆栈深度

#进入超级用户模式，并禁止 FIQ 中断和 IRQ 中断
MSR     CPSR_c, #Mode_SVC|I_Bit|F_Bit
#紧接着 IRQ 模式下的堆栈，设置超级用户下的栈顶指针
MOV     SP, R0
SUB     R0, R0, #SVC_Stack_Size      #超级用户下的堆栈深度

#设置进入用户模式
MSR     CPSR_c, #Mode_USR
#紧接着超级用户模式下的堆栈，设置用户模式下的栈顶指针，剩余的空间都开辟为堆栈
MOV     SP, R0

MOV     PC, R1           #堆栈初始化子程序返回

#重映射 SRAM 区
RemapSRAM:

MOV     R0, #0x40000000   //RAM 区首地址
LDR     R1, =Vectors      //向量表首地址

```


#下面一段程序是把从 0x00000000 开始的 64 字节（Flash 中的中断向量表和地址表）搬移到以 #0x40000000 为首地址的 RAM 区中

R9 中	LDMIA R1!, {R2-R9}	//把以[R1]为首地址的 32 字节数据装载到 R2~
元中	STMIA R0!, {R2-R9}	//把 R2~R9 中的数据存入以[R0]为首地址的单
R9 中	LDMIA R1!, {R2-R9}	//把以[R1]为首地址的 32 字节数据装载到 R2~
元中	STMIA R0!, {R2-R9}	//把 R2~R9 中的数据存入以[R0]为首地址的单

#下面几行代码设置存储器映射控制寄存器

	LDR R0, =MEMMAP	//取 MEMMAP 地址到 R0
	MOV R1, #0x02	
RAM 区重新映射	STR R1, [R0]	//给 MEMMAP 赋值为 0x02，设置中断向量从
	mov pc, lr	//跳转到主程序

#下面一段程序代码是进入软中断来切换系统的工作模式，当希望从一种模式切换到另一种模式时，可以通过调用下面对应标号的程序段进入软中断。在软中断处理程序中会根据所给定的中断号处理，执行 SWI #num 后软中断号被存入 R0 中

```
.globl disable_IRQ
.globl restore_IRQ
.globl ToSys
.globl ToUser
```

#禁止 IRQ

disable_IRQ:

STMFD SP!, {LR}	//把 LR 值压入堆栈
swi #0	//产生 0 号软中断，0 → R0
LDMFD SP!, {pc}	//恢复 PC 值，返回

#恢复 IRQ

restore_IRQ:

STMFD SP!, {LR}	//把 LR 值压入堆栈
swi #1	//产生 1 号软中断，1 → R0
LDMFD SP!, {pc}	//恢复 PC 值，返回

#进入系统工作模式

ToSys:

STMFD SP!, {LR}	//把 LR 值压入堆栈
swi #11	//产生 11 号软中断，11 → R0

```

        LDMFD    SP!, {pc}                //恢复 PC 值, 返回

#进入用户工作模式

ToUser:
        STMFDP   SP!, {LR}                //把 LR 值压入堆栈
        swi      #12                      //产生 12 号软中断, 11→R0
        LDMFD    SP!, {pc}                //恢复 PC 值, 返回

#软中断处理代码
SWI_Exception:
        STMFDP   SP!, {R2-R3,LR}          //把 R2、R3、LR 的值压入堆栈

#0 号软中断的处理程序
CMP     R0, #0                            //将 R0 和 0 比较
        //以下 4 行带 EQ 条件的代码均为当 R0 为 0 时应该执行的语句
        MRSEQ    R2, SPSR                 //把 SPSR 读入 R2 中
        STREQ    R2, [R1]                 //把 R2 的值存入 [R1]中
        ORREQ    R2, R2, #0x80            //将 R2 的 Bit7 位置 1
        MSREQ    SPSR_c, R2               //把 R2 的值写入 SPSR_c 中, 即禁止 IRQ

#1 号软中断的处理程序
CMP     R0, #1                            //比较 R0 值和 1
        LDREQ    R2, [R1]                 //相等则把[R1]中的数据存入 R2 中
        MSREQ    SPSR_c, R2               //相等则把 R2 的值写入 SPSR_c 中, 恢复 IRQ

#11 号软中断的处理程序
CMP     R0, #11                           //比较 R0 的值和 11
        MRSEQ    R2, SPSR                 //相等则把 SPSR 的值转存入 R2 中
        BICEQ    R2, R2, #0x1F            //相等则把 R2 的 Bit0~Bit4 全部清零
        ORREQ    R2, R2, #Mode_SYS        //相等则把 R2 和#Mode_SYS 相与再存入 R2
        MSREQ    SPSR_c, R2               //相等则把 R2 的值存入 SPSR_c 中, 即进入

系统模式

#12 号软中断的处理程序
CMP     R0, #12                           //比较 R0 的值和 12
        MRSEQ    R2, SPSR                 //相等则把 SPSR 的值存入 R2
        BICEQ    R2, R2, #0x1F            //相等则把 R2 的 Bit0~Bit4 清零
        ORREQ    R2, R2, #Mode_USR        //相等则把 R2 和#Mode_USR 相与再存入

R2 中

        MSREQ    SPSR_c, R2               //相等则把 R2 存入 SPSR_c 中, 即进入用

户模式

        LDMFD    SP!, {R2-R3,PC}          //恢复 R2、R3、PC 的值, 返回

        .END                             //汇编代码段结束

```

4.5 从 ADS 1.2 到 Realview MDK

Keil 是业界最好的 51 单片机开发工具之一，它拥有流畅的用户界面与强大的仿真功能。ARM 将 Keil 公司收购之后，正式推出了针对 ARM 微控制器的开发工具 RVMDK，它将 ARM 编译器 RVCT 与 Keil 的工程管理、调试仿真工具集成在一起，是一款非常强大的 ARM 微控制器开发工具。

很多嵌入式系统开发工程师对 ARM 的老版本开发工具 ADS 1.2 非常熟悉，而 RVMDK 与 ADS 相比较，在外观、仿真流程及内部二进制编译链接工具上都有了不少改进，用法稍有不同。下面介绍通用的流程及一些注意事项，帮助 ADS 1.2 用户将老的、遗留的 ADS 1.2 工程转化成在 RVMDK 上进行开发调试的工程。

4.5.1 工具结构的改进

ARM 新推出的微控制器开发工具 RVMDK 与 ADS 1.2 在工具架构组成上有一些不同，这些区别包括：不同版本的 ARM 编译器（Compiler），不同的调试器（Debugger），不同的仿真器（Simulator），以及不同的硬件调试单元。作为 ARM 的新一代微控制器开发工具，RVMDK 不但包含 ARM 的最新版本编译链接工具，即 RVDS 3.0 编译链接工具，而且根据微控制器调试开发的特点采用了与 ADS、RVDS 完全不同的调试、仿真环境——uVision Debugger 与 Simulator。

RVMDK 集成了 RVDS 3.0 的编译工具 RVCT 3.0，与 ADS 1.2 相比，除去编译、链接工具的可执行二进制文件不同之外，RVCT 3.0 的很多编译链接选项与 ADS 编译器也有不同。

1. POSIX 格式

RVCT 3.0 采用 POSIX 格式的编译链接选项，所有的多字符选项前必须使用双短线。例如，ADS 的编译选项 `-cpu`，在 RVMDK 中需要改写成 `--cpu`，否则用户在 RVMDK 中直接使用 ADS 的 makefile 时工具会产生如下警告：

```
Warning: L3910W: Old syntax, please use '--xxx'
```

2. 编译器例化形式

在 ADS 中，当用户要将高级语言代码编译成目标文件时，需要根据目标机器码的不同（16 位的 Thumb 代码或者 32 位的 ARM 代码），以及高级语言的不同（C 代码或者 C++ 代码），选择不同的编译器可执行文件；RVCT 3.0 编译器则将它们全部统一为 `armcc`，仅仅通过不同的编译选项进行区分。表 4-3 较为详细地列出了其中的差别。

表 4-3 RVMDK 与 ADS 编译器的例化形式对比

ADS 1.2	RVMDK 3.0	默认的编译选项
Armcc	armcc	-c90 -arm
Tcc	armcc -thumb	-c90
armcpp	armcc -cpp	-arm
Tcpp	armcc -thumb -cpp	

注意：表 4-3 中“默认的编译选项”是指在没有其他编译选项时编译器的默认选项。

3. 连接器的使用

对目标文件进行链接之前，ARM 工具的连接器会严格检查各个文件（objects），判断它们是否符合 ARM 体系结构的 ABI 标准。由于 RVCT 与 ADS 编译链接工具所遵循的 ARM ABI 不同，所以将 ADS 的遗留工程直接移植到 RVMDK 并进行连接时，用户可能会遇到如下的错误或者警告：

```
Error: L6238E: foo.o(.text) contains invalid call from '~PRES8' function to 'REQ8' function
Warning: L6306W: '~PRES8' section foo.o(.text) should not use the address of 'REQ8' function foobar
```

这是因为新工具的 ABI 要求在函数调用时，系统必须保证堆栈指针 8byte 对齐，即每次进栈或者出栈的寄存器数目必须为偶数。这是为了能够更加高效地使用 STM 与 LDR 指令对“double”或“long long”类型的数据进行访问。而老的 ARM 开发工具 ADS 并没有考虑到新的 ARM 内核架构，其 ABI 对于堆栈的操作仅仅要求 4byte 对齐。所以当用户将在 ADS 中编译连接成功的工程代码移植到 RVMDK 上，或者将老的、ADS 遗留的目标文件、库文件在新工具 RVMDK 中进行连接时，RVMDK 的连接器就会报出以上的错误。

对于以上情况，用户可以通过简单修改代码并重新编译链接，或者使用特殊的编译选项来解决。

4. 重新编译所有代码

当用户拥有该 ADS 遗留工程的所有源代码时，使用 RVMDK 重新编译链接全部代码是最好的解决方法。RVMDK 中的新版本编译工具会重新生成满足堆栈 8byte 对齐要求的目标文件，避免由于堆栈不对齐引起的连接错误。

当工程中包含汇编代码时，用户可能还需要做少量的代码修改。这些修改包括以下内容。

（1）检查汇编源码中的指令，确保堆栈操作指令是 8byte 对齐的。

如 Ex.1 中，ADS 的遗留代码一次性将 5 个寄存器压栈，由于 ARM 的指令寄存器宽度为 32 位，即 4byte，显然 5 个寄存器入栈之后，堆栈指针不能满足 64 位、8byte 对齐。为了解决这一问题，可以将另外一个并不需要压栈的寄存器 R12 同时压栈，这样当 6 个 32 位寄存器进栈之后，堆栈就能满足 64 位对齐了。

Ex.1

```
STMFD sp!, {r0-r3, lr}           ; 将 R0、R1、R2、R3、LR（奇数）寄存器压入堆栈
```



```
STMFD sp!, {r0-r3, r12, lr}      ; 将偶数个寄存器压入堆栈
```

（2）在每个汇编文件的开头添加“PRESERVE8”指令，详见 Ex.2。

Ex.2

```
AREA Init, CODE, READONLY
```



```
PRESERVE8
AREA Init, CODE, READONLY
```

5. 使用--apcs/adsabi 编译选项

当用户没有该 ADS 遗留工程的全部源码，只拥有库文件或者目标文件时，可以通过--apcs/adsabi 编译选项强制 RVMDK 的编译器产生符合 ADS ABI 要求的目标文件，以达到与遗留的 ADS 库文件、目标文件兼容的目的。

注意：ARM 新工具将不会继续支持--apcs/adsabi 选项。建议及时更新工具到最新版本。

4.5.2 分散加载文件

RVMDK 同样支持 ADS 的分散加载文件，但是当分散加载文件中涉及必须被放置 ROOT Region 中的 C 库函数时，用户需要进行少量修改。

ROOT Region 的 load address 与 execution address 相同，所以这部分代码在系统初始化时无须进行搬移操作。很多库函数，如__scatter*.o 和__dc*.o，必须被放置在 Root Region 中。

Ex.3—分散加载文件的修改

; ADS 中的分散加载文件

```
ROM_LOAD 0x0
{
    ROM_EXEC 0x0
    {
        vectors.o (Vect, +First)
        __main.o (+RO)
        * (Region$$Table)
        * (ZISection$$Table)
    }
    RAM_EXEC 0x100000
    {
        *.o (+RO,+RW,+ZI)
    }
}
```



; RVMDK 中的分散加载文件 1

```
ROM_LOAD 0x0
{
    ROM_EXEC 0x0
    {
        vectors.o (Vect, +First)
        * (InRoot$$Sections)
    }
    RAM_EXEC 0x100000
    {
        *.o (+RO,+RW,+ZI)
    }
}
```

; RVMDK 中的分散加载文件 2

```
ROM_LOAD 0x0
{
    ROM_EXEC 0x0
    {
        vectors.o (Vect, +First)
        __main.o(*)
        * (Region$$Table)
        __scatter*.o(*)
        __dc*.o(*)
    }
    RAM_EXEC 0x100000
    { *.o (+RO,+RW,+ZI) }
}
```

在 ADS 中, 用户必须在分散加载文件中明确地将特定 Section 代码放置在 Root Region 中。而 RVMDK 为了支持新的 RW 压缩机制, 采用了新的 Region Table 格式, 这种新的格式并不包含 ZISection\$\$Table, 而且新的 scatter-loading (__scatter*.o) 与 decompressor (__dc*.o) 必须被放置在 Root Region 中。所以 Ex.3 中 ADS 的分散加载文件应该被修改成新的形式。Ex.3 中提供了两种修改分散加载文件的方法, 分散加载文件 1 通过 InRoot\$\$Sections 自动将所有必需的库目标放置在 Root Region 中, 而分散加载文件 2 则详细地注明了 __scatter*.o 与 __dc*.o 的位置。

4.5.3 C 库函数的差异

为了与新的 ABI 一致, RVMDK 中的库函数名称与 ADS 可能会有不同。ADS 中的 __rt_* 库函数被替换为 __aeabi_*. 如果用户的 ADS 工程中曾经修改 (retarget) 过这些库函数, 那么在移植到 RVMDK 时, 需要重新实现这些函数, 以满足新 ABI 的要求。表 4-4 列出了部分函数的对应关系。

表 4-4 部分库函数的对应关系

ADS 库函数	RVMDK 库函数
__rt_memcpy_w	__aeabi_memcpy4
__rt_div0	__aeabi_idiv0
__rt_sdiv	__aeabi_idiv
__rt_udiv	__aeabi_udiv
__rt_fp_status_addr	__aeabi_fp_status_addr
__rt_erno_addr	__aeabi_erno_addr

4.5.4 开发环境迁移实例

结合以上对 RVMDK 与 ADS 差异的描述, 本节将以实例的形式叙述如何将 ADS 1.2 的遗留代码移植到 RVMDK 中。

以 PHILIPS 的 LPC 2294 (ARM 7 TDMI) 为处理器, 将一个在 ADS 1.2 上开发的由 LPC 2294 控制 LED 闪烁的工程移植到 RVMDK 上来。由图 4-27 可以看出, 该工程 (Legacy_ADS.mcp) 共有 4 个源文件 (Startup.s、target.c、IRQ.s、main.c), 以及一个分散加载文件 (Scatterload)。

使用 ADS 1.2 编译器。

编译选项为: -O1 -g+。

链接选项为: -info totals -entry 0x00000000 -scatter .\src\Scatterload.scf -info sizes。

得到最终代码尺寸信息如下:

Total RO	Size(Code + RO Data)	1640 (1.60kB)
Total RW	Size(RW Data + ZI Data)	1128 (1.10kB)
Total ROM	Size(Code + RO Data + RW Data)	1640 (1.60kB)

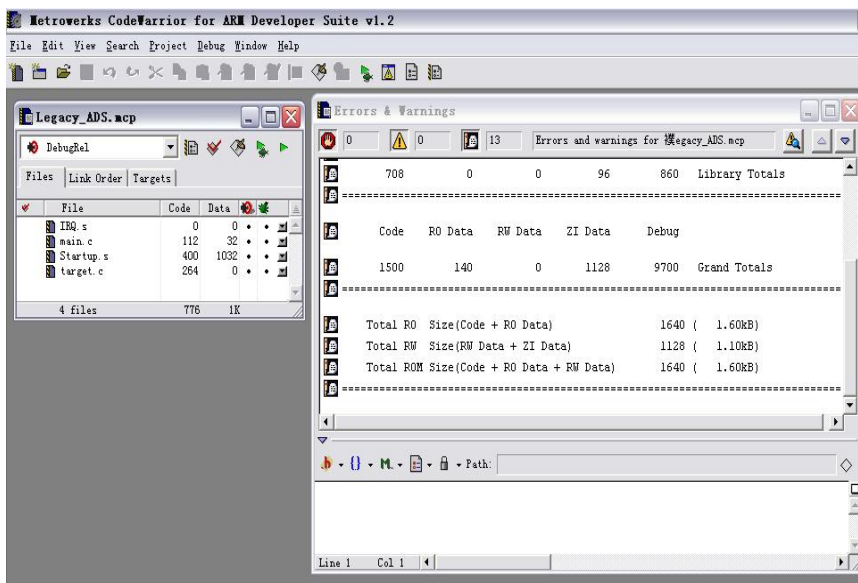


图 4-27 ADS 遗留工程

为了能够使用 ARM 新工具 RVMDK 的一系列特性，需要把 ADS 中的遗留工程移植到 RVMDK 上来。具体步骤如下。

1) 在 RVMDK 中新建工程

打开 RVMDK，在主菜单中选择“Project”→“New...”→“uVision Project”，并将新工程命名为“New_MDK.uv2”，单击“保存”按钮，如图 4-28 所示。

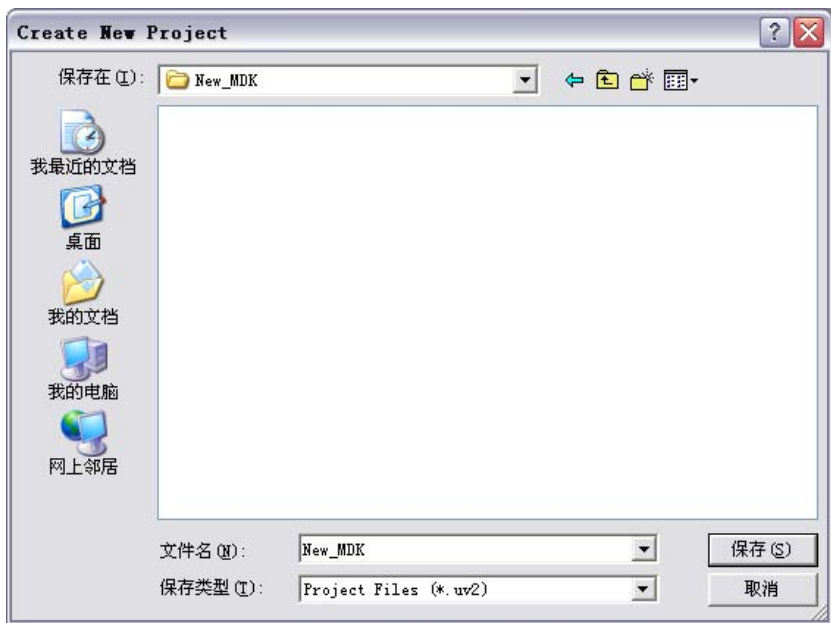


图 4-28 在 RVMDK 中新建工程

在 RVMDK 自动弹出的器件选择窗口（Select Device for Target）中选择该工程所对应的处理器型号“LPC2294”，并单击“确定”按钮，如图 4-29 所示。RVMDK 提示用户是否自动添加启动代码时，选择“否”。

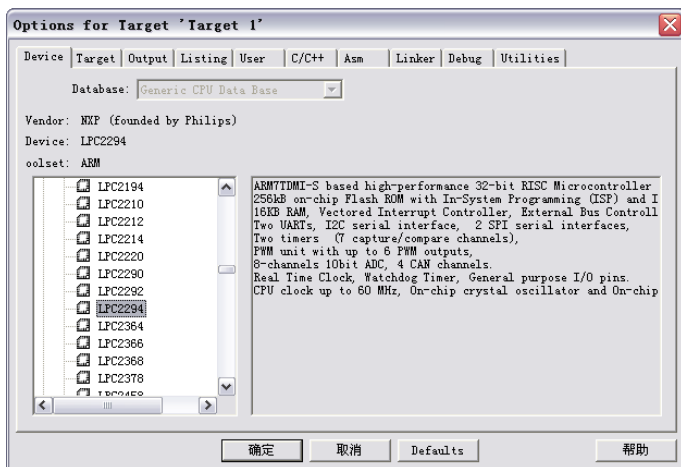


图 4-29 在 RVMDK 中选择合适的处理器

2) 添加源文件并设置工程属性

将 Legacy_ADS.mcp 工程中所有的源文件都添加到新的 New_MDK.uv2 工程中，如图 4-30 所示。

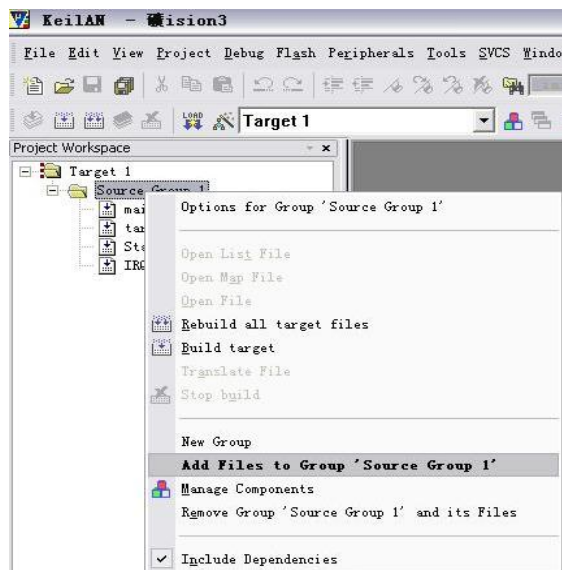



图 4-30 将 ADS 工程的遗留源代码全部添加到新工程中

单击工程属性快捷按钮 ，打开工程属性设置窗口，选择“C/C++”标签页，设置编译器属性。用户可以根据以前 ADS 工程的编译属性设置，也可以根据当前的具体需求重新设

置编译属性。在本例中，将 ADS 遗留工程的编译属性“-O1 -g+”修改为“-O1 -g -W”后，复制到“Misc Controls”栏中，如图 4-31 所示。这是因为由于编译器版本的变化，其对应的编译选项也有所变化的缘故。

注意：“-W”选项可以抑制所有的 warning。

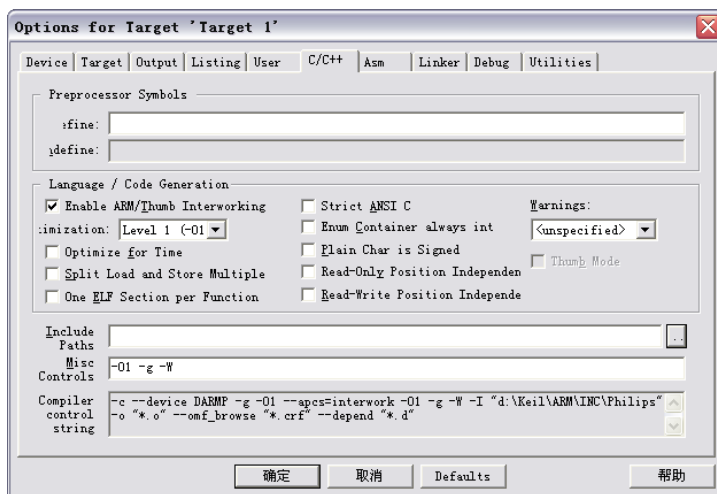


图 4-31 编译选项的设置

对 ADS 工程中的链接选项作适当修改，如下所示，使其符合 POSIX 格式。

```
--info totals --entry 0x00000000 --scatter .\src\Scatterload.scf --info sizes
```

选择“Linker”标签页，将修改过的链接选项复制到 RVMDK 工程属性的 Linker 属性中，并单击“确定”按钮，如图 4-32 所示。

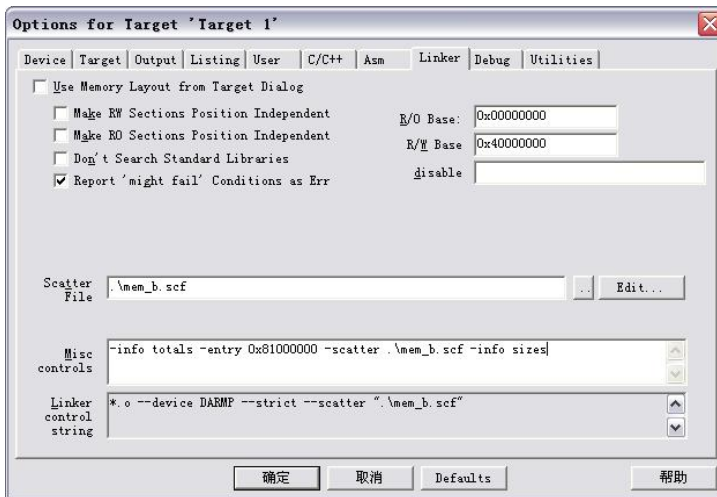



图 4-32 链接选项的设置

3) Build 工程并适当修改代码

当所有的工程属性都设置好之后,单击“Build all target file”快捷按钮,对整个工程进行编译链接。在 RVMDK 窗口的 Build 输出一栏中,会发现系统出现了一个链接错误、“L6238E”,如图 4-33 所示,这是由于 RVMDK 中新版本编译链接工具与 ADS 的老版本 Build 工具采用不同的 ABI 造成的。

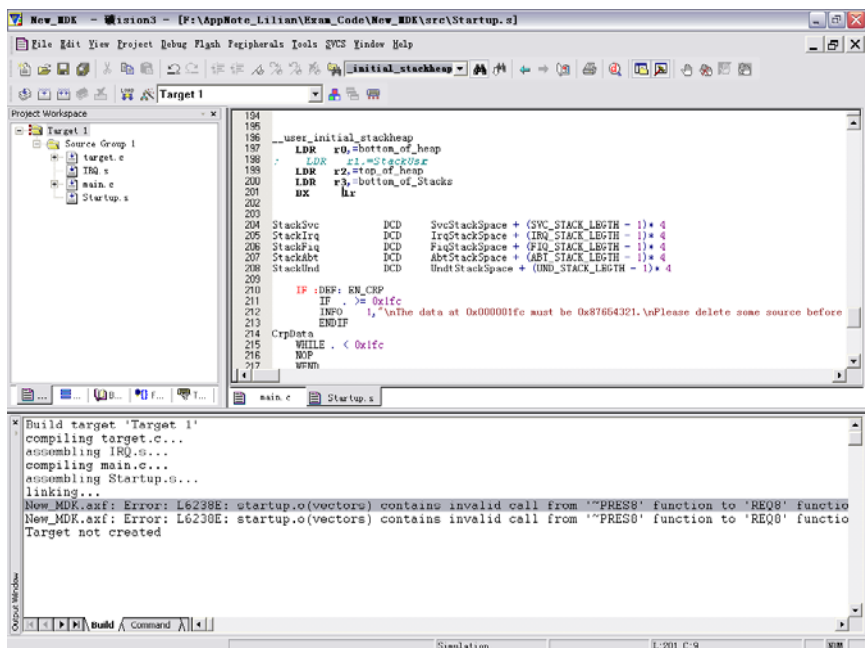



图 4-33 链接错误“L6238E”

为此,打开该工程中的汇编文件 Startup.s,在该程序第 55 行添加 PRESERVE8 指令,如下所示:

```
CODE32
PRESERVE8
AREA    vectors,CODE,READONLY
```

4) 重新编译链接该工程

代码修改完毕之后,单击“Build all target file”快捷按钮,对该工程进行二次编译链接。RVMDK 将成功生成 New_MDK.axf 文件,并显示其代码尺寸信息为:

```
Program Size: Code=1576  RO-data=64 RW-data=0 ZI-data=1128
```

5) 工程调试

与其他 ARM 开发工具相比较,RVMDK 拥有非常出色的仿真功能,可以帮助用户在纯软件的平台上进行较为精确的调试。用户可以在工程属性设置窗口选择 Simulator 调试(如图 4-34 所示),或者通过硬件调试工具(uLink)进行调试。

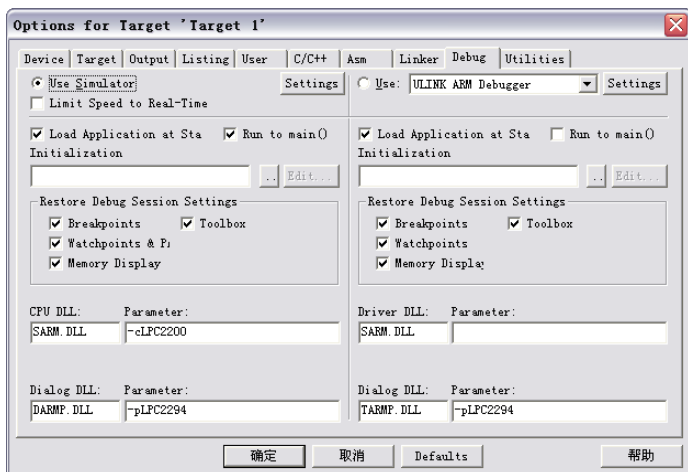



图 4-34 选择 uVision Simulator 作为调试平台

当选择 Simulator 调试时，单击 Debug 快捷按钮，打开 Simulator 调试窗口，如图 4-35 所示。为了验证该程序是否能在 LPC 2294 硬件平台上正确执行，可通过 GPIO 口驱动 LED 进行循环闪烁，用户可以单击 Peripherals→GPIO→Port2，将 GPIO 端口 2 的仿真界面打开。

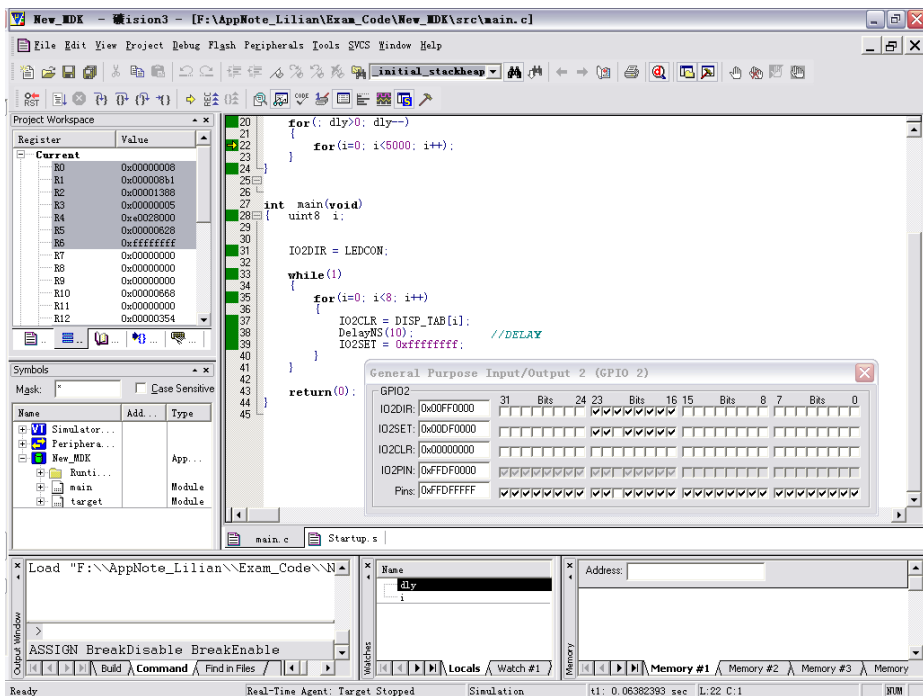


图 4-35 Simulator 调试窗口

单击运行快捷按钮，可以看到在 GPIO 端口 2 的仿真调试窗口中，I/O 口的输出在不停地循环变化。

第 5 章

ARM 常用外设接口

在设计 ARM 控制系统以前，首先要了解常用的外设。ARM 处理器内部集成了很多的片内外设控制器，大大减少了外部硬件数量。常用的外设控制器有 I²C、SPI、I²S、LCD、网络 PHI，有些还是有 SD 卡、CF 卡的硬件控制器。当然，在内部没有提供硬件控制器的情况下，在有些要求不高的场合可以用软件模拟引脚的时序来控制外部设备，本章开头部分将涉及。对于接口引脚多并且信号变化复杂的场合，也可以用 CPLD 或者 FPGA 来做控制器，按照外设接口的时序，用 VHDL 代码实现，同时在 FPGA 内部设计一系列与接口相关的寄存器，ARM 通过寄存器和 FPGA 交互，而 FPGA 通过 VHDL 代码生成的逻辑和外设硬件交互，达到 ARM 和外设交互的目的。

5.1 SPI 接口

串行外围设备接口即 SPI，是 Serial Peripheral Interface 的缩写，由 Motorola 首先在其 MC68HCXX 系列处理器上定义。SPI 接口主要应用在 EEPROM、FLASH、实时时钟、A/D 转换器，以及数字信号处理器和数字信号解码器之间。SPI 是一种高速的全双工同步通信总线，并且在芯片的引脚上只占用 4 根线，节约了芯片的引脚，同时为 PCB 的布局节省空间，提供方便。正是出于这种简单易用的特性，现在越来越多的芯片集成了这种通信协议。

SPI 总线系统可与各个厂家生产的多种标准外围器件直接接口，该接口一般使用 4 条线：串行时钟线 SCK、主机输入/从机输出数据线 MISO、主机输出/从机输入数据线 MOSI 和低电平有效的从机选择线 SS（有的 SPI 接口芯片带有中断信号线 INT 或 INT $\bar{}$ ，有的 SPI 接口芯片没有主机输出/从机输入数据线 MOSI）。

SPI 的通信原理很简单，它以主从方式工作，如图 5-1 所示。这种模式通常有一个主设备和一个或多个从设备，需要至少 4 根线，事实上 3 根也可以（单向传输时）。这也是所有基于 SPI 的设备共有的，它们是 SDI（数据输入）、SDO（数据输出）、SCK（时钟）、CS（片选）。

- SDO——主设备数据输出，从设备数据输入。
- SDI——主设备数据输入，从设备数据输出。
- SCLK——时钟信号，由主设备产生。
- CS——从设备使能信号，由主设备控制。

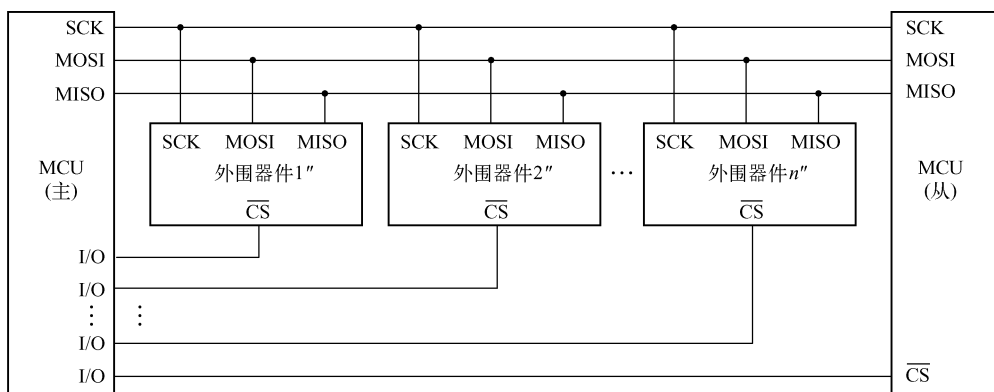


图 5-1 SPI 的主从结构

CS 是控制芯片是否被选中的，也就是说只有片选信号为预先规定的使能信号时（高电位或低电位），对此芯片的操作才有效。这就使得在同一总线上连接多个 SPI 设备成为可能。

接下来是负责通信的 3 根线。通信是通过数据交换完成的，这里先要知道 SPI 是串行通信协议，也就是说数据是一位一位传输的。这是 SCK 时钟线存在的原因，即由 SCK 提供时

钟脉冲, SDI、SDO 基于此脉冲完成数据传输。数据输出通过 SDO 线, 数据在时钟上升沿或下降沿时改变, 在紧接着的下降沿或上升沿被读取。完成一位数据传输, 输入也使用同样原理。这样, 在至少 8 次时钟信号的改变中 (上升沿和下降沿为一次), 就可以完成 8 位数据的传输。

需要注意的是, SCK 信号线只由主设备控制, 从设备不能控制该信号线。同样, 在一个基于 SPI 的设备中, 至少有一个主控设备。这样的传输方式有一个优点, 与普通的串行通信不同, 普通的串行通信一次连续传送至少 8 位数据, 而 SPI 允许数据一位一位地传送, 甚至允许暂停, 因为 SCK 时钟线由主控设备控制, 当没有时钟跳变时, 从设备不采集或传送数据。也就是说, 主设备通过对 SCK 时钟线的控制可以完成对通信的控制。SPI 还是一个数据交换协议: 因为 SPI 的数据输入和输出线独立, 所以允许同时完成数据的输入和输出。不同 SPI 设备的实现方式不尽相同, 主要是数据改变和采集的时间不同, 在时钟信号上升沿或下降沿采集有不同的定义, 具体请参考相关器件的文档。

在点对点的通信中, SPI 接口不需要进行寻址操作, 并且为全双工通信, 显得简单高效。在多个从设备的系统中, 每个从设备需要独立的使能信号, 硬件上比 I²C 系统要稍微复杂一些。

5.2 模块式 LCD 的 SPI 接口设计

一般的 ARM 应用系统都带有液晶显示, 对于简单的 ARM 7 或者 ARM Cortex 工业控制器, 并没有内置液晶控制器外设, 需要外加一个液晶模块, 处理器和液晶之间的通信采用 I/O 的方式进行。下面首先介绍液晶接口的一些基本知识, 然后提供实例讲解如何接口和编程。

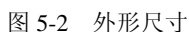
液晶显示器由于能够提供友好的用户交互界面, 越来越多的嵌入式产品开始采用液晶显示器。用菜单方式的人机交互大大方便了用户的使用, 用户只需在菜单的提示下操作, 而不需要像以前那样记住烦琐的控制设置。

5.2.1 128×64 点阵显示器

目前, 带有中文字库的 128×64 点阵的液晶模块使用量大、价格便宜。RT12864M 汉字图形点阵液晶显示模块, 可显示汉字及图形, 内置 8 192 个中文汉字 (16×16 点阵)、128 个字符 (8×16 点阵) 及 64×256 点阵显示 RAM (GDRAM)。其主要的技术指标如下。

- 电源: VDD 3.3~+V (内置升压电路, 无须负压)。
- 显示内容: 128 列×64 行。
- 显示颜色: 黄绿。
- 显示角度: 6:00 钟直视。
- LCD 类型: STN。
- 与 MCU 接口: 8 位或 4 位并行/3 位串行。
- 配置 LED 背光。
- 多种软件功能: 光标显示、画面移位、自定义字符、睡眠模式等。

- 其外形尺寸如图 5-2 所示。



128×64 点阵液晶模块的引脚说明见表 5-1。

• 152 •

续表

引脚号	引脚名称	方向	功能说明
11	DB4	H/L	数据 4
12	DB5	H/L	数据 5
13	DB6	H/L	数据 6
14	DB7	H/L	数据 7
15	PSB	H/L	并/串行接口选择：H—并行；L—串行
16	NC	—	空脚
17	/RET	H/L	复位，低电平有效
18	NC	—	空脚
19	LED_A	—	背光源正极（LED+5V）
20	LED_K	—	背光源负极（LED-OV）

从表 5-1 中可以看出，LCD 模块有两种工作方式：串行和并行。由于并行方式占用的口线比较多，因此这里选择串行方式。串行方式遵守 SPI 协议，而 ARM 带有 SPI 接口，因此可直接挂在 SPI 总线上面。图 5-3 为串行数据传送格式。

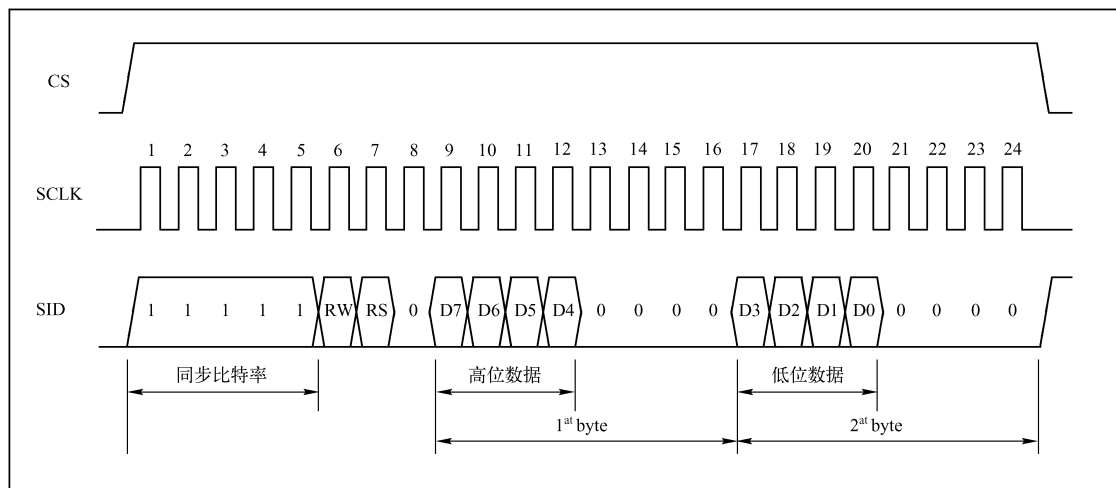


图 5-3 LCD 模块串行数据传送格式

串行数据传送分为 3 字节完成。

第一字节：串口控制。格式：11111ABC。

A 为数据传送方向控制：H 表示数据从 LCD 到 MCU，L 表示数据从 MCU 到 LCD。

B 为数据类型选择：H 表示数据是显示数据，L 表示数据是控制指令。

C 固定为 0。

第二字节：（并行）8 位数据的高 4 位。格式：DDDD0000。

第三字节：（并行）8 位数据的低 4 位。格式：0000DDDD。

到目前为止，我们对 LCD 的接口已经有了基本的了解。现在开始设计和 ARM 处理器的接口。接口需要如下口线：

- SPI 数据入；
- SPI 数据出；
- SPI 时钟线；
- 片选线；
- LCD 复位线；
- 模块电源线；
- 背光电源线；
- 对比度电压调节。

5.2.3 模块式 LCD 硬件连接

通过 SPI 方式和处理器的硬件连接如图 5-4 所示。图中由于应用场合的需要，液晶的背光一直打开。液晶模块的背光采用 4 个高亮的发光二极管，因此一直打开消耗的电流达 80mA。

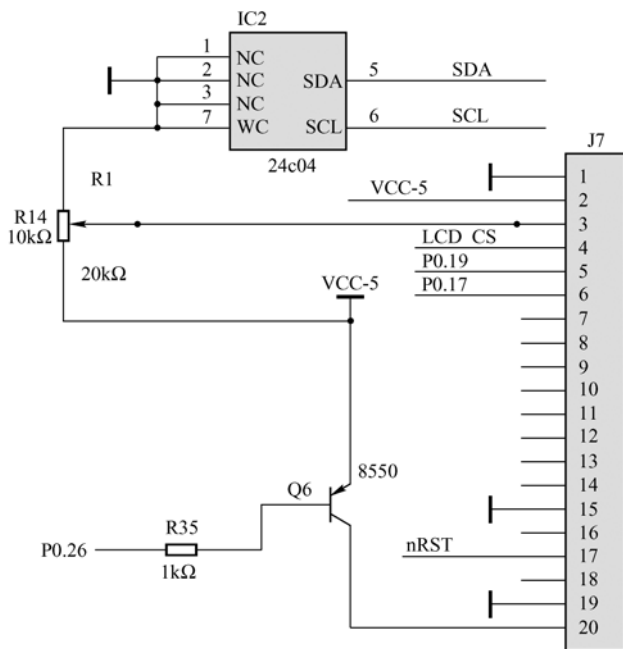
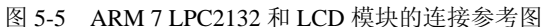


图 5-4 通过 SPI 方式和处理器的硬件连接

图 5-5 是一个完整的 ARM 芯片和 LCD 接口的原理图，编程的思路需要按照原理图的连接方式来展开。这里是 SPI 方式，因此最底层的接口函数是通过 SPI 传输数据的。如果用并行方式，则要修改最底层的接口函数，在此函数上层的其他程序代码都是一样的。



5.2.4 模块式 LCD 内部存储器

硬件连接完成后，下面介绍代码的编制。软件的编制分为两个主要部分：液晶模块的接口程序，菜单的 API 函数接口。模块的接口程序编制需要参考液晶的技术规格说明书。

模块采用 ST7920A 控制芯片，由于模块可以显示文字和图形，因此需要了解如何对这两种工作方式进行操作。其实，模块内部是通过文本显示 RAM 和绘图 RAM 来实现的。程序代码需要参考两个指令集合：基本指令集和扩充指令集。在编写驱动之前，先来了解内部显示 RAM 的操作。

1. 文本显示 RAM (DDRAM)

文本显示 RAM 提供 8 个×4 行的汉字空间，当写入文本显示 RAM 时，可以分别显示 CGROM、HCGROM 与 CGRAM 的字型；ST7920A 可以显示三种字型，分别是半宽的 HCGROM 字型、CGRAM 字型及中文 CGROM 字型。三种字型的选择，由在 DDRAM 中写入的编码确定，各种字型的详细编码如下。

- 显示半宽字型：将 1 字节写入 DDRAM 中，范围为 02H~7FH 的编码。
- 显示 CGRAM 字型：将 2 字节编码写入 DDRAM 中，总共有 0000H、0002H、0004H、0006H 4 种编码。
- 显示中文字型：将 2 字节编码写入 DDRAMK 中，范围为 A1A0H~F7FFH (GB 码) 或 A140H~D75FH (BIG5 码) 的编码。

2. 绘图 RAM (GDRAM)

绘图显示 RAM 提供 128×8 字节的记忆空间。在更改绘图 RAM 时，先连续写入水平与垂直的坐标值，再写入 2 字节的数据，地址计数器 (AC) 会自动加 1；在写入绘图 RAM 期间，绘图显示必须关闭。整个写入绘图 RAM 的步骤如下：

(1) 关闭绘图显示功能。

(2) 将水平的位元组坐标 (x) 写入绘图 RAM 地址；将垂直的坐标 (y) 写入绘图 RAM 地址；将 D15~D8 写入 RAM 中；将 D7~D0 写入 RAM 中；打开绘图显示功能。

(3) 游标/闪烁控制。

ST7920A 提供硬件游标及闪烁控制电路，由地址计数器 (Address Counter) 的值来指定 DDRAM 中的游标或闪烁位置。

对液晶编程还涉及定位问题，包括文字和图形，因此需要参照此款液晶的定位安排址，如图 5-6 所示。

1) 图形显示坐标

水平方向 x——以字节为单位；

垂直方向 y——以位为单位。

2) 汉字显示坐标 (见表 5-2)

表 5-2 汉字显示坐标

	x 坐标							
Line1	80H	81H	82H	83H	84H	85H	86H	87H
Line2	90H	91H	92H	93H	94H	95H	96H	97H
Line3	88H	89H	8AH	8BH	8CH	8DH	8EH	8FH
Line4	98H	99H	9AH	9BH	9CH	9DH	9EH	9FH

由图 5-6 可以发现，图形点阵分为上、下两个屏：图形的 x （字节）坐标的变化范围是 0~15，但不是在同行连续的，而是 0~7 字节在上半屏 y 坐标对应的一行，8~15 字节在下半屏 y 坐标对应的一行，编程的时候需要注意。文字显示也一样，80H 位置是第一个汉字的位置，但 88H 放在了下半屏。

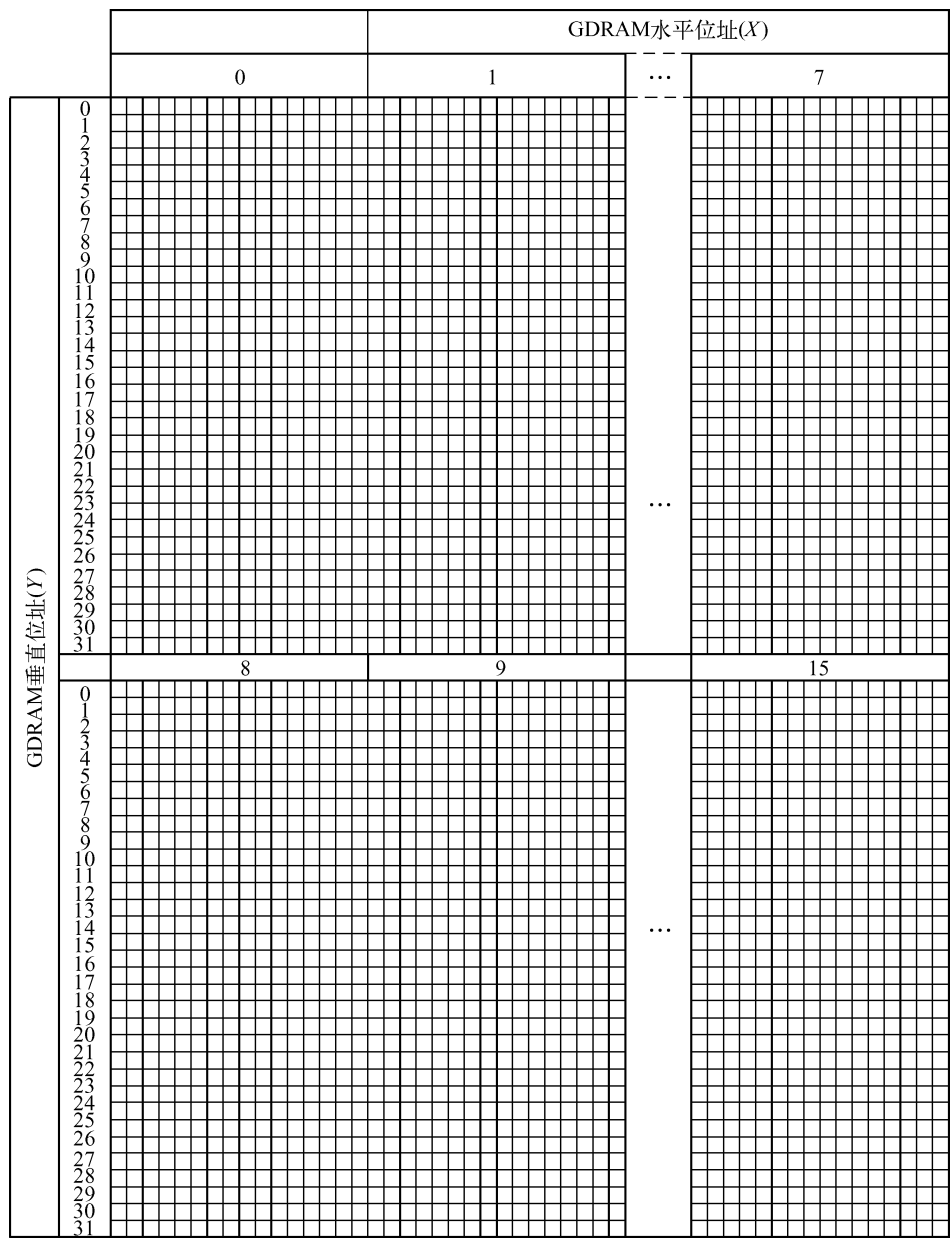


图 5-6 控制器内部位址

5.2.5 SPI 接口 LCD 显示程序

下面的程序是对 LCD 进行类似于手机的菜单式显示的函数集，包括文字和图片的显示，由于比较简单，请按照注释自行分析。

```

/* File: LCDDISP.C
* 功能：使用硬件 SPI 接口输出 LCD 显示
*****/

#include "config.h"
#include "function.h"
unsigned char const picture[64][16]={
//图片数据太大，此处省略
}
short displaybuffer[64][8]; //定义显示缓冲区
extern void delay(uint32 dly);
void Blink(int row); //对某一行文字反白显示
void TextDisplayInit(void); //显示初始化函数
void DisplayPicture(void); //显示图片函数
/*****
* 名称：MSpiIni()
* 功能：初始化 SPI 接口
* 入口参数：无
* 出口参数：无
*****/

void MSpiIni(void)
{ S0PCCR= 52; //设置 SPI 时钟分频
  S0PCR = 0x30; //设置 SPI 接口模式，MSTR=1, CPOL=1, CPHA=0, LSBF=0
}

/*****
* 名称：MSendData()
* 功能：向 SPI 总线发送数据
* 入口参数：data，待发送的数据
* 出口参数：返回值为读取的数据
*****/

void MSendData(char data)
{ S0PDR = data;
  while( 0==(S0PSR&0x80) ); //等待 SPIF 置位，即等待数据发送完毕
}

/*****
* 名称：SendLCD()
* 功能：向 LCD 发送数据或命令
* 入口参数：data，待发送的数据。命令/数据选择：1——命令；0——数据
*****/

```

```

void SendLcd(uint8 data,uint8 flag)
{
char firstbyte,secondbyte;
MSpiIni();                //初始化 SPI 接口
firstbyte=data&0xf0;
secondbyte=data<<4;
secondbyte=secondbyte&0xf0;
IODIR|=LCD_CS;
IOSET|= LCD_CS;           //片选
if(flag==1)
MSendData(controlcommand); //发送命令
if(flag==0)
MSendData(displaycommand);
MSendData(firstbyte);      //发送数据
MSendData(secondbyte);     //发送数据
IOCLR|=LCD_CS;
}

/*****
* 名称: PrintSentence()
* 功能: 向显示器打印句子
* 入口参数: x——开始的行, 本例的液晶采用 4 行 8 列汉字;
*           p——指向句子的指针
*****/
void PrintSentence(unsigned short int x,char *p)
{
unsigned int chineseword;    //存放一个汉字的变量
unsigned short int temp1,temp2,y;
//LCD 初始化开始
SendLcd(0x06,1);           //DDRAM 的地址计数器 (AC) 加 1
//LCD 初始化结束
for( y=0;y<8;y++)          //打印 1 行汉字
{
temp1=(unsigned short int)*p++;
temp2=(unsigned short int)*p++;
chineseword=temp1;
chineseword=(chineseword<<8)|(unsigned int)temp2; //获取汉字代码, 汉字为 2 字节表示
DisplayChineseword(x,y, chineseword);             //在坐标 x,y 处显示汉字
}
}

/*****
* 名称: DisplayChineseword()
* 功能: 在(x,y)位置的一个汉字显示

```

```

* 入口参数(x,y)坐标, 存放汉字的变量 chineseword
*****/

void DisplayChineseWord(unsigned short int x,unsigned short int y,unsigned int chineseWord)
{
    unsigned int i;
    char i1,i2;
    i=(unsigned int)chineseWord;
    i2=i;
    i1=i>>8;
    switch(x){
        case 0:
            SendLcd(0x80+y,1);
            break;
        case 1:
            SendLcd(0x90+y,1);
            break;
        case 2:
            SendLcd(0x88+y,1);
            break;
        case 3:
            SendLcd(0x98+y,1);
            break;
    }
    SendLcd(i1,0);
    SendLcd(i2,0);
}

/*****
*此函数负责将屏幕的(x,y)坐标放到对应的显示缓冲区里
*****/

void PrintDot(int x,int y)
{
    displaybuffer[x][y/16]=0x8000>>(y%16);
}

/*****
* 名称: ClearPicture()
* 功能: 清除 LCD 图形函数
*****/

void ClearPicture()
{
    int i,j;
    SendLcd(0x32,1);
    SendLcd(0x36,1);
    SendLcd(0x34,1);
}

```

```

        for(i=0;i<64;i++)
            for(j=0;j<8;j++)
                displaybuffer[i][j]=0;
    }

    /**
     * 名称: ClearSentence()
     * 功能: 清除 LCD 所有文字函数
     */
    void ClearSentence( ){
        SendLcd(0x30,1);           //功能设置——8bit 控制界面，基本指令集
        SendLcd(0x0c,1);           //显示打开，光标关，反白显示关
        SendLcd(0x01,1);           //清除屏幕显示，将 DDRAM 的地址计数器归零
    }

    /**
     * 函数名称: PrintXY()
     * 函数功能: 在(x,y)处打点
     */
    void PrintXY(int x,int y,int Bold)
    {
        if(x<0) x=0;
        if(y<0) y=0;
        if(x>127) x=127;
        if(y>63) y=63;
        if(Bold==0)
            PrintDot(64-y,x);
        else {PrintDot(64-y,x);
            PrintDot(64-y+1,x);
        }
    }

    /**
     * 功能: 从(x,y)处，向 y 方向画线，线长为 length
     */
    void Line_Y(int startX,int startY,int length)
    {
        int i;
        for (i=startX;i<length;i++)
            PrintDot(i,startY);
    }

    /**
     * 功能: 从(x,y)处，向 x 方向画线，线长为 length
     */

```



```

/*****/

void Line_X(int startX,int startY,int length)
{
    int i;
    SendLcd(0x34,1);                //关显示
    for (i=startY;i<length;i++)
        PrintDot(startX,i);
    SendLcd(0x36,1);                //功能设置——8bit 控制界面，扩充指令集
}

void GraphShow()                    //在 LCD 输出显示 buffer 的内容
{
    int i,j;
    SendLcd(0x30,1);                //基本命令
    SendLcd(0x02,1);                //地址归位
    SendLcd(0x36,1);                //8bit 控制界面，扩充指令集
    SendLcd(0x32,1);                //功能设置——8bit 控制界面，绘图显示 ON
    SendLcd(0x34,1);                //关显示
    for(i=0;i<64;i++)
        for(j=0;j<8;j++)
        {
            SendLcd(0x80+i%32,1);    //设置绘图区的 y 地址坐标
            SendLcd(0x80+(j+((int)(i/32))*8)%128,1); //设置绘图区的 x 地址坐标
            SendLcd(displaybuffer[i][j]>>8,0);    //显示缓冲区的数据输出
            SendLcd(displaybuffer[i][j],0);
        }
    SendLcd(0x36,1);                //开显示
}

void TextDisplayInit()
{
    SendLcd(0x30,1);                //功能设置——8bit 控制界面，基本指令集
    SendLcd(0x0c,1);                //显示打开，光标关，反白显示关
    SendLcd(0x01,1);                //清除屏幕显示，将 DDRAM 的地址计数器归零
    SendLcd(0x02,1);
}

/*****/
* 功能：显示一幅图片
* 将事先保存的图片数据读到显示缓冲区里
/*****/

void DisplayPicture( )
{
    int i,j,temp1,temp2;
    for(i=0;i<64;i++)

```

```

        for(j=0;j<8;j++)
        {
            temp1 =0x00;
            temp2=temp1|picture[i][j*2];
            temp2=temp2<<8;
            temp1=temp2|picture[i][j*2+1];
            displaybuffer[i][j]=temp1;
        }
    }

//反白显示
void Blink(int row)
{int i,j;

// SendLcd(0x32,1);                //功能设置——8bit 控制界面，绘图显示 ON
    SendLcd(0x34,1);                //关显示

    for(i=0;i<64;i++)
        for(j=0;j<8;j++)
        {
            SendLcd(0x80+i%32,1);    //设置绘图区的 y 地址坐标

            SendLcd(0x80+j+((int)(i/32))*8,1);    //设置绘图区的 x 地址坐标

            if((row*16<i)&&(i<(row+1)*16) ){ SendLcd(0xff,0);
                SendLcd(0xff,0);
            }
            else{
                SendLcd(0x0,0);
                SendLcd(0x0,0);
            };
            SendLcd(0x36,1);        //功能设置——8bit 控制界面，扩充指令集

        }

    }

/*****
* 函数功能：某个文字处光标闪烁
* 入口参数：location 代表文字的位置
*****/

```

```
void Flash(int location){
    if(location!=0)
    {
        SendLcd(0x30,1);        //基本指令
        //SendLcd(0x,1);
        SendLcd(0xf,1);        //光标开
        SendLcd(0x7,1);
        SendLcd(0x80+location,1);
    }
}
```

嵌入式系统中 SPI 接口用了很多，特别是 SPI 的存储器应用非常广泛，这里介绍 SPI 的存储器接口编程。

5.3 SPI Flash Memory 编程

对程序员而言，Flash 编程是一件费力的事情，特别是串行总线的 Flash。其实原理都大同小异，中央处理器给出需要读取或写入的地址及对应的数据，Flash 找到相应的地址，取出或者存入对应的数据。与普通的存储器不同的是，Flash 在写入时需要更多的步骤。写 Flash 的复杂性体现在以下几个方面：

(1) 写以前必须先将对对应位置的数据擦除。如果不擦除，则新写入的数据将和老的数据产生结合，从而使写入结果不正确。例如，假设写操作只能使片内的位由 1 变为 0。空白片子所有的位都是 1，第一次在首字节写入 11111110，结果没有问题，第二次写入 11111111，由于原先最低位是 0，现在如果不执行擦除操作就没有办法使其变为 1，因此结果还是 11111110，从而使写入数据不正确。

(2) 一次只能擦除一个 Sector 或者 Block，而不能擦除单个字节。根据器件的不同，每个 Sector 的大小不一样。

(3) 不同的器件擦除和写入数据的流程不同，

5.3.1 SPI Flash 硬件接口

下面以 ATMEL 公司的 4MB Flash AT45DB041 为例进行介绍。它和 ARM 7 的中央处理器 LPC2114 的接口如图 5-7 所示。

AT45DB041 的内存体系分为三个层次：SECTOR、BLOCK 及 PAGE。PAGE 是最小的单位，每一页包含 264 字节。这里采用页索引来定位页地址。页也是写入、读出数据和擦除的最小单位。该驱动程序将做成 API 函数，主要包括：WriteMemory——写一页数据；ReadMemory——读一页数据；StreamWrite——页数据顺序写入（不用关心当前页索引）；GetIndex——获取当前页索引。以上这些函数实现一般的数据存储足足有余，但没有考虑容错。例如，认为存储器没有问题，写入的数据没有校验。增加容错将降低写入的速度。对于一般的工业应用而言，上述函数已经足够。

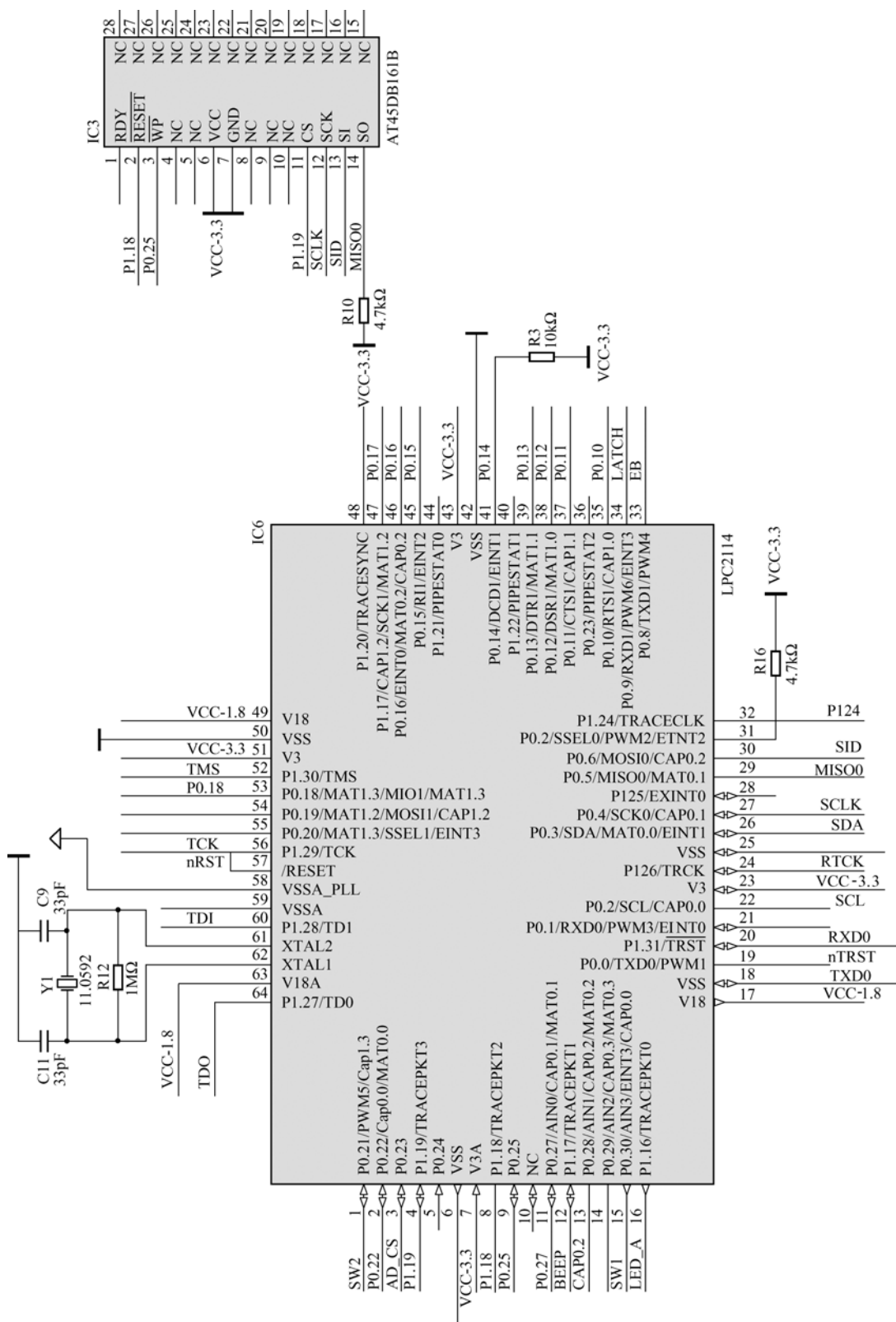


图 5-7 AT45DB041 串行 SPI Flash 和 ARM 的连接

5.3.2 AT45DB041 的软件接口函数

本节是一个项目中对 SPI Flash 操作的实例，内部包含了基本的操作函数集。根据其实现可以发现，程序的编制要严格按照芯片的手册进行。一般步骤是先看懂芯片的 Datasheet，然后开始写函数，最后在线调试。调试中要注意 SPI 发出的波形和数据是否正常，延迟是否正确，只有时序满足要求了，芯片才能正常工作。因此调试的时候需要数字示波器或者逻辑分析仪的辅助。

```
#include "function.h"
#include "config.h"
#include "misc.h"
#define FLASH_SIZE    264
#define BUFFER_1_READ 0xd4
#define BUFFER_2_READ 0xd6
#define BUFFER_1_WRITE 0x84
#define BUFFER_2_WRITE 0x87

int bottom;
void delay(int);
char *memorybuffer;
char str[FLASH_SIZE];

/*****
 *
 * Function: AT45SPIinit()
 * Description: Initialize the Flash.
 * Notes: This function includes flash memory chip selection,write protect
 *        disable and disable LCD chip selection to avoid conflict.
 * Returns: none
 *****/
void AT45SPIinit()
{
    S0PCCR=8;
    S0PCR =0x30;    //CPOL=1, CPHA=0, MSTR=1, LSBF=0
    IO1DIR|=AT45DB_CS;//I/O 方向控制设置
    IO0DIR|=AT45DB_WP;//I/O 方向控制设置
    IO1CLR|=AT45DB_CS;//片选
    IO0SET|=AT45DB_WP;//去掉 Flash 写保护
    IO1SET|=AT45DB_CS; //deselect memory
    IO0DIR|=LCD_CS;    //I/O 方向控制设置
    IO0CLR|=LCD_CS;    //disable LCD chip selection
}

/*****
 * Function: MReadData()
```

```

*
* Description: only generate SCK of SPI for reading operation.
* Returns: none
*****/

char MReadData(){
    MSendData(0x0); //dummy command, only generate sck
    return S0PDR ;
}

/*****
* Function: MSendData()
* Description: 向 SPI 总线发送数据
* Returns: 返回值为读取的数据
*****/

void MSendData(uint8 data)
{
    S0PDR = data;
    while( 0==(S0PSR&0x80) ); //等待 SPIF 置位, 即等待数据发送完毕
}

/*****
* Function: WriteMemory()
* Description: Write data to page in the Flash.
* Notes: This function is specific to the AT45DBxxx Flash memory.
* Returns: none.
*****/

void WriteMemory(char *buff,int pageindex){
    //memory index 包括页及页内缓冲区
    int length;
    char temp1,temp2,temp3;
    AT45SPInit(); //初始化 SPI 接口
    temp1=pageindex>>7;
    temp2=pageindex<<1;
    temp3=0;
    IO1CLR|=AT45DB_CS; //select memory
    MSendData(0x82); //main memory page program through buffer1
    MSendData(temp1);
    MSendData(temp2); //
    MSendData(temp3); //9 address bits
    for(length=0;length<FLASH_SIZE;length++)
        MSendData(*(buff+length)); //memory write
    IO1SET|=AT45DB_CS; //deselect memory
    delay(230);
}

```

```

IO1CLR|=AT45DB_CS;//select memory
MSendData(0xd7);//memory status
while((MReadData()&0x80)!=0x80);
IO1SET|=AT45DB_CS; //deselect memory
}
/*****

* Function: ReadMemory()
* Description: Read one page data from Flash.
* Notes: This function is specific to the AT45DBxxx Flash memory.
*
* Returns: none
*****/

//data stored in buff,each time one page will be read out
void ReadMemory(char *buff,int pageindex){
    char temp1,temp2,temp3;
    int length=0;
    AT45SPInit();           //初始化 SPI 接口
    temp1=pageindex>>7;
    temp2=pageindex<<1;
    temp3=0;
    IO1SET|=AT45DB_CS;
    delay(10);
    IO1CLR|=AT45DB_CS;//select memory
    MSendData(0x55);//main memory to buffer 2 transfer
    MSendData(temp1);
    MSendData(temp2);
    MSendData(temp3);
    IO1SET|=AT45DB_CS; //deselect memory
    IO1CLR|=AT45DB_CS;//select memory
    while((MReadData()&0x80)!=0x80);
    IO1SET|=AT45DB_CS; //deselect memory
    delay(1000);
    IO1CLR|=AT45DB_CS;//select memory
    //buffer 2 read
    MSendData(BUFFER_2_READ);
    MSendData(0x0);
    MSendData(0x0);
    MSendData(0x0);
    MSendData(0x0);
    for(length=0;length<FLASH_SIZE;length++)
        *(buff+length)=MReadData();    //read data
    IOSET|=AT45DB_CS; //deselect memory

}
/*****

* Function: StreamWrite()

```

* Description: write one page data to Flash,the application programmer needn't consider the current pageindex.

*

* Notes: This function is specific to the AT45DBxxx Flash memory.

* Returns: none

*****/

```
void StreamWrite(char *buff)
{
    int pageindex;
    int *p;
    char st[FLASH_SIZE];
    char temp1;
    p=(int *)st;
    AT45SPInit();           //初始化 SPI 接口
    IO1SET|=AT45DB_CS; //deselect memory
    IO1CLR|=AT45DB_CS; //select memory
    MSendData(0xd7); //get memory status
    while((MReadData() & 0x80) != 0x80);
    //获取芯片型号 4MB、8MB
    temp1=MReadData();
    switch((temp1 >> 2) & 0x0f)
    {
        case 7://4MB
            bottom=2048; //2048 page each page has FLASH_SIZE bytes
            break;
        case 9://8MB
            bottom=4096;
            break;

        default:
            bottom=2048;
    }
    IO1SET|=AT45DB_CS; //deselect memory
    pageindex=bottom/4; //电压、电流首页地址
    ReadMemory((char *)st,bottom-1);
    pageindex=*p; //获取存储的 pageindex
    if(pageindex <= bottom-2)
        pageindex++; //increase pageindex
    else pageindex=bottom/4;
    *p=pageindex;
    WriteMemory((char *)st,bottom-1); //存储 pageindex
    WriteMemory(buff,pageindex); //store data
}
```



```

/*****
* Function: GetIndex()
* Description: Get the latest pageindex of Flash.
* Notes: This function is specific to the AT45DBxxx Flash memory.
* Returns: pageindex
*****/
int GetIndex()//根据时间查找。0: 查到数据 index; 1: 故障 index
{
    char *buf,temp1;
    int *p,pageindex;
    char str[FLASH_SIZE];
    buf=str;
    AT45SPInit();           //初始化 SPI 接口
    IO1CLR|=AT45DB_CS;//select memory
    MSendData(0xd7);//get memory status
    while((MReadData()&0x80)!=0x80);
    //获取芯片型号 4MB、8MB
    temp1=MReadData();
    switch((temp1>>2)&0x0f)
    {
        case 7://4MB
            bottom=2048; //2048 page each page has FLASH_SIZE bytes
            break;
        case 9://8MB
            bottom=4096;
            break;
        default:
            bottom=2048;
    }
    IO1SET|=AT45DB_CS; //deselect memory
    ReadMemory((char *)str,bottom-1);
    p=(int *)str;//获取存储的 pageindex
    pageindex=*p;
    return pageindex;
}

```

5.4 I²C 接口

I²C (Inter-Integrated Circuit) 总线是同步通信的一种特殊形式，具有接口线少、控制方式简单、器件封装小、通信速率较高等优点。它通过串行数据 (SDA) 和串行时钟 (SCL) 线在连接到总线的器件间传递信息。每个器件都有一个唯一的地址识别，而且都可以作为发送器或接收器 (由器件的功能决定)。除了发送器和接收器外，器件在执行数据传

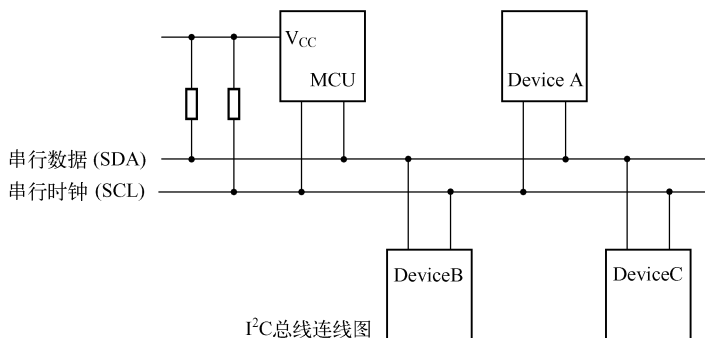
输时也可以被看作主机或从机（见表 5-3）。主机是初始化总线的数据传输并产生允许传输的时钟信号的器件，此时，任何被寻址的器件都被认为是从机。

表 5-3 I²C 总线定义

术 语	描 述
发送器	发送数据到总线的器件
接收器	从总线接收数据的器件
主机	初始化发送、产生时钟信号和终止发送的器件
从机	被主机寻址的器件
多主机	同时有多于一个主机尝试控制总线，但不破坏报文
仲裁	是一个在有多个主机同时尝试控制总线，但只允许其中一个控制总线并使报文不被破坏的过程
同步	两个或多个器件同步时钟信号的过程

5.4.1 上拉电阻与传输速率

I²C 总线的 SCL 和 SDA 端口输出为漏极开路，因此使用时必须连接上拉电阻。如图 5-8 所示为 I²C 多机通信。

图 5-8 I²C 多机通信

上拉电阻的大小与电源电压、传输速率等有关。这是由于外围器件是漏极开路，上拉电阻的大小在一定程度上影响外围器件信号传输的延迟时间。

串行的 8 位双向数据传输位速率如下：

- 标准模式下可达 100kb/s，采用 10k Ω 的上拉电阻。
- 快速模式下可达 400kb/s，采用 2k Ω 的上拉电阻。
- 高速模式下可达 3.4Mb/s。高速模式下不执行仲裁和时钟同步以加速位处理能力，同时高速模式主器件有一个 SDAH 信号的开漏输出缓冲器和一个在 SCLH 输出的开漏极下拉和电流源上拉电路，这个电流源电路缩短了 SCLH 信号的上升时间。任何时候在高速模式下，只有一个主机的电流源有效。

I²C 总线上的外围扩展器件为电压型负载的 CMOS 器件，因此总线上的器件数量不是由

电流负载能力决定的，而是由电容负载能力决定的。通常 I^2C 总线的负载能力为 400PF。同时由于外围器件的地址唯一，所以外围器件的数量还受器件地址空间的限制。

5.4.2 I^2C 总线三种信号

I^2C 总线在传送数据过程中共有三种信号，它们分别是启动信号、停止信号和应答信号。

- 启动信号：SCL 为高电平时，SDA 由高电平向低电平跳变，开始传送数据。
- 停止信号：SCL 为低电平时，SDA 由低电平向高电平跳变，结束传送数据。
- 应答信号：接收数据的 IC 在接收到 8b 数据后，向发送数据的 IC 发出特定的低电平脉冲，表示已收到数据。CPU 向受控单元发出一个信号后，等待受控单元发出一个应答信号，CPU 接收到应答信号后，根据实际情况做出是否继续传递信号的判断。若未收到应答信号，则判断为受控单元出现故障。

1. I^2C 位传输

I^2C 串行总线有两根信号线：一根双向的数据线 SDA，一根时钟线 SCL。所有接到 I^2C 总线上的设备的串行数据都接到总线的 SDA 线，各设备的时钟线 SCL 接到总线的 SCL。

- 数据传输：SCL 为高电平时，SDA 线若保持稳定，则 SDA 上是在传输数据 bit；若 SDA 发生跳变，则用来表示一个会话的开始或结束。
- 数据改变：SCL 为低电平时，SDA 线才能改变传输的 bit。

位传输波形如图 5-9 所示。

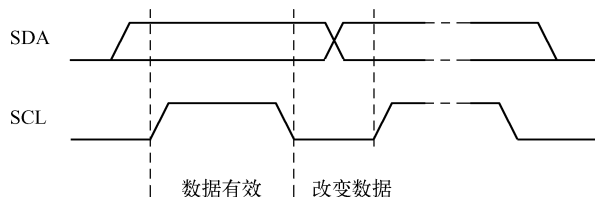


图 5-9 I^2C 位传输波形

2. I^2C 开始和结束信号

- 开始信号：SCL 为高电平时，SDA 由高电平向低电平跳变，开始传送数据。
- 结束信号：SCL 为高电平时，SDA 由低电平向高电平跳变，结束传送数据。

开始和结束信号如图 5-10 所示。

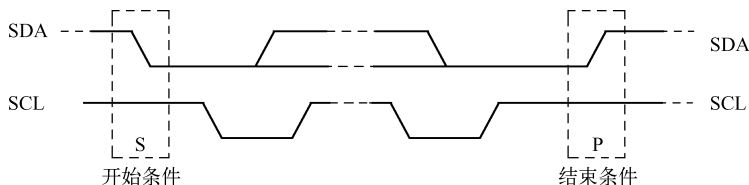


图 5-10 I^2C 开始和结束信号

3. I²C 应答信号

主机每发送完 8b 数据后等待从机的 ACK。即在第 9 个 Clock，若从 IC 发出 ACK，则 SDA 会被拉低。

若没有 ACK，SDA 会被置高，这会引起主机发生 RESTART 或 STOP 流程，如图 5-11 所示。

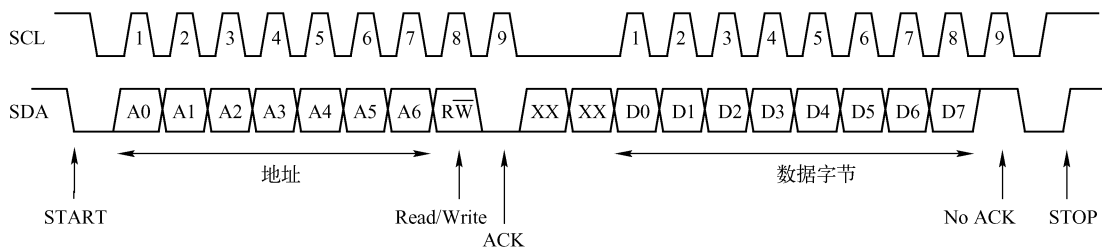


图 5-11 I²C 的应答波形

4. I²C 写流程

写寄存器的标准流程如下：

- (1) 主机发起 START；
- (2) 主机发送 I²C addr (7b) 和 W 操作 0 (1b)，等待 ACK；
- (3) 从机发送 ACK；
- (4) 主机发送 reg addr (8b)，等待 ACK；
- (5) 从机发送 ACK；
- (6) 主机发送 data (8b)，即要写入寄存器中的数据，等待 ACK；
- (7) 从机发送 ACK；
- (8) 第 (6) 步和第 (7) 步可以重复多次，即顺序写多个寄存器；
- (9) 主机发起 STOP。

写一个寄存器的波形如图 5-12 所示。

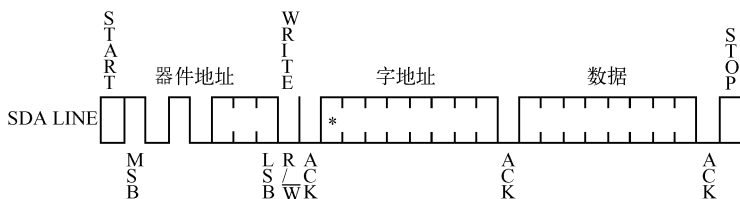


图 5-12 I²C 写单个寄存器

写多个寄存器的波形如图 5-13 所示。

5. I²C 读流程

读寄存器的标准流程如下：

- (1) 主机发送 I²C addr (7b) 和 w 操作 1 (1b)，等待 ACK；
- (2) 从机发送 ACK；

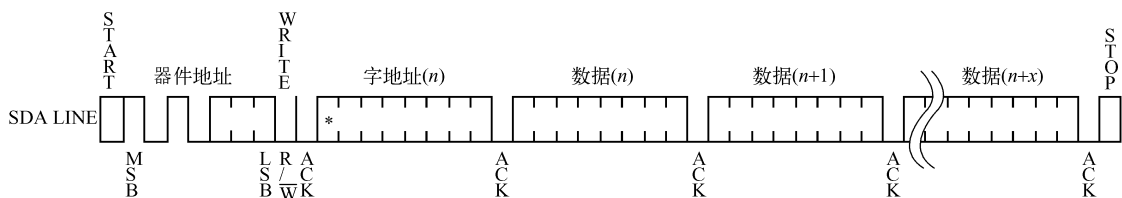


图 5-13 I²C 写多个寄存器波形

- (3) 主机发送 reg addr (8b), 等待 ACK;
 - (4) 从机发送 ACK;
 - (5) 主机发起 START;
 - (6) 主机发送 I²C addr (7b) 和 r 操作 1 (1b), 等待 ACK;
 - (7) 从机发送 ACK;
 - (8) 从机发送 data (8b), 即寄存器中的值;
 - (9) 主机发送 ACK;
 - (10) 第 (8) 步和第 (9) 步可以重复多次, 即顺序读多个寄存器。
- 读一个寄存器的波形如图 5-14 所示。

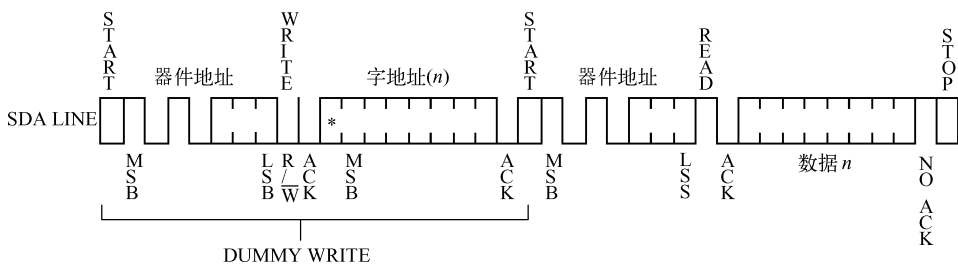


图 5-14 I²C 读一个寄存器的波形

读多个寄存器的波形如图 5-15 所示。

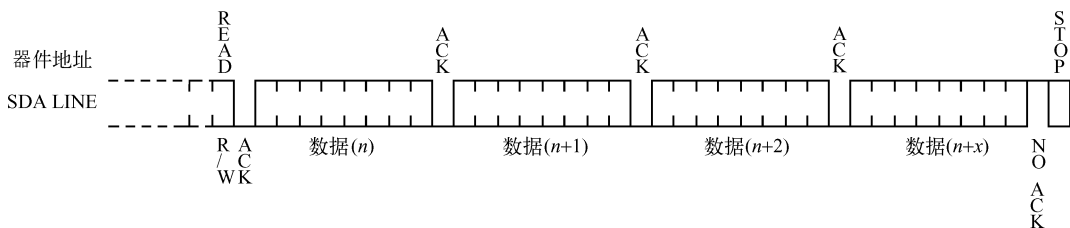


图 5-15 I²C 读多个寄存器的波形

5.4.3 软件仿真 I²C 示例

如果芯片不带 I²C 硬件控制器, 也可以用软件的方式模拟, 只要时序符合 I²C 芯片的波形要求即可。为了说明问题, 假设某款 ARM 处理器的硬件 I²C 接口已经接了其他 I²C 从设

备，现在还需要再接一个 I²C 从设备，并且这个设备希望单独和处理器引脚连接，这时 I²C 硬件接口不够用，可以用 I/O 模拟方式实现 I²C。以下程序是最简单的模拟 I²C 程序，对 AT24C02 进行操作。

定义 I²C 器件地址，器件是 7 位地址，参考图 5-12 和图 5-14，读和写对应的最后一位是不同的。

```

/*****
#define WriteDeviceAddress 0xa0
#define ReadDviceAddress 0xa1

*****/

```

定义 SDA 高电平为 SDA1，低电平为 SDA0。

定义 SCL 高电平为 SCL1，低电平为 SCL0。

```

#define SDA1  rPDATF|=0x2;
#define SDA0  rPDATF&=0xfd;
#define SCL1  rPDATF|=0x1;
#define SCL0  rPDATF&=0xfe;

```

短延迟函数：

```

void delay(int i){
    for(;i>0;i--);
}

```

长延时函数：

```

/*****
void DelayMs(unsigned int number) {
    int i;
    for(i=0;i<number*10000;i++)
        ;
}
*****/

```

按照 I²C 启动时序编写，参考图 5-10。

```

/*****
void Start() {
    SDA1;
    delay(100);
    SCL1;
    delay(100);
    SDA0;
    delay(100);
    SCL0;
}
*****/

```

```
    delay(100);
}
```

按照 I²C 结束时序编写，参考图 5-10。

```
/******  
void Stop() {  
    SCL0;  
    delay(100);  
    SDA0;  
    delay(100);  
    SCL1;  
    delay(100);  
    SDA1;  
    delay(100);  
}
```

应答信号的软件模拟，参考图 5-11。

```
/******  
void Ack() {  
    SDA0;  
    delay(100);  
    SCL1;  
    delay(100);  
    SCL0;  
    delay(100);  
    SDA1;  
    delay(100);  
}
```

NOACK 参考图 5-11。

```
/******  
void NoAck() {  
    SDA1;  
    delay(100);  
    SCL1;  
    delay(100);  
    SCL0;  
    delay(100);  
}
```

查询 ACK 应答：

```
/******  
char TestAck() {  
    char ErrorBit;
```

```

SDA1;
delay(100);
SCL1;
rPCONF&=~0xc;
while((rPDATF&0x2)!=0);
rPCONF|=0x4;
SCL0;
return(ErrorBit);
}

```

写 8b 的子程序，在读、写 AT24C02 时都会用到：

```

/*****
char Write8Bit(unsigned char input) {
    unsigned char temp,i;
    for(temp=8;temp!=0;temp--) {

        if(input&0x80){
            SDA1;}
        else { SDA0;

        }

        delay(100);
        SCL1;
        delay(100);
        SCL0;
        input=input<<1;
    }
}
*****/

```

向 AT24C02 连续写多字节的数据：

```

/*****
void Write24c02(unsigned char *Wdata,unsigned char RomAddress,unsigned char number) {
    int i;
    Start();
    Write8Bit(WriteDeviceAddress);
    TestAck();
    Write8Bit(RomAddress);
    TestAck();
    for(i=0;i<number;i++) {

        Write8Bit(*Wdata);
        TestAck();
        Wdata++;
    }
}
*****/

```



```

    }
    Stop();
    DelayMs(10);
}

```

读 8b:

```

/*****
unsigned char Read8Bit() {
    unsigned char temp,rbyte=0;
    rPCONF&=~0xc;
    for(temp=8;temp!=0;temp--) {
        SCL1;
        rbyte=rbyte<<1;

        rbyte=rbyte|((unsigned char)((rPDATF&0x2)?1:0));
        SCL0;
        delay(100);
    }
    rPCONF|=0x4;
    return(rbyte);
}
*****/

```

从 AT24C02 连续读多字节的数据:

```

/*****
void Read24c02(unsigned char *RamAddress,unsigned char RomAddress,unsigned char bytes) {
// unsigned char temp,rbyte;
    Start();
    Write8Bit(WriteDeviceAddress);
    TestAck();
    Write8Bit(RomAddress);
    TestAck();
    Start();
    Write8Bit(ReadDviceAddress);
    TestAck();
    while(bytes!=1) {
        *RamAddress=Read8Bit();
        Ack();
        RamAddress++;
        bytes--;
    }
    *RamAddress=Read8Bit();
    NoAck();
    Stop();
}
*****/

```

5.4.4 带 I²C 硬件控制器的程序

以下程序是 LPC2114 的 ARM 对 AT24C02 进行操作，因为处理器带硬件控制器，所以不需要软件模拟，可以直接对硬件寄存器进行设置，由硬件自动完成 I²C 的传输操作，和 AT24C02 进行交互。当然，初次可能会认为没有模拟方式的容易理解，不过只要知道其 I²C 的初始化和传输都是通过硬件寄存器完成的，也就容易理解了。

```
//24c02 器件读/写地址定义
#define ReadAddr 0xa1
#define WriteAddr 0xa0
//I2C 控制置位寄存器 I2CONSET 位定义
#define I2EN 0x40 //I2C 接口使能位
#define STA 0x20 //起始标志位
#define STO 0x10 //停止标志位
#define SI 0x08 //中断标志位
#define AA 0x04 //产生应答位

//I2C 接口初始化
void I2CInit(void){
    PINSEL0=0x00000050; //P0.2/P0.3 引脚连接 SCL/SDA 功能
    I2SCLH=10; //设置 I2C 频率
    I2SCLL=10;
    I2CONSET=I2EN; //使能 I2C 接口
}

//向 24C02 指定的地址写 1 字节
void writeByte(uchar8 addr,uchar8 data){
    I2CONSET=STA; //设置控制寄存器，发送起始条件
    I2DAT=addr; //通过寄存器 I2DAT，硬件自动送出地址
    I2CONCLR=SI; //读取应答
    while(!(I2CONSET & SI));
    I2DAT=WriteAddr; //通过寄存器 I2DAT，写器件地址
    I2CONCLR=SI; //读取应答
    while(!(I2CONSET & SI));

    I2DAT=data; //通过寄存器 I2DAT，送出数据
    I2CONCLR=SI; //读取应答
    while(!(I2CONSET & SI));

    I2CONSET=STO; //通过寄存器 I2CONSET，发送结束条件
}

//从 24C02 指定的地址读取 1 字节
uchar8 readByte(uchar8 addr){
    uchar8 data;

    I2CONSET=STA; //通过寄存器 I2CONSET，发送起始条件
```

```
I2CONCLR=SI; //读取应答
while(!(I2CONSET & SI));

I2DAT=WriteAddr; //通过寄存器 I2DAT, 写器件地址
I2CONCLR=SI; //读取应答
while(!(I2CONSET & SI));

I2DAT=addr; //通过寄存器 I2DAT, 送出地址
I2CONCLR=SI; //读取应答
while(!(I2CONSET & SI));
I2CONSET=STA; //发送起始条件
I2CONCLR=SI; //读取应答
while(!(I2CONSET & SI));

I2DAT=ReadAddr; //读器件地址
I2CONCLR=SI; //读取应答
while(!(I2CONSET & SI));
I2CONCLR=SI; //读取应答
while(!(I2CONSET & SI));
data=I2DAT; //读取数据
I2CONSET=STO; //发送停止条件
return data;
}
//延时函数
void delay(){
    int i;
    for(i=0;i<1000;i++)
        ;
}
int main(void){
    uchar8 index,temp;
    uchar8 str[]={ "hello world" };
    I2CInit(); //初始化 I2C 控制器
    //写入数据
    for(index=0;index<11;index++){
        writeByte(index,str[index]);
        delay();
    }
    //读取数据
    for(index=0;index<11;index++)
        temp=readByte(index);
    while(1);
}
```

第 6 章

基于 STM32 的室内导航家用拖地机

从本章开始进入基于 ARM 的产品开发，ARM 的开发包括不含操作系统的裸机程序开发和带操作系统的开发。带操作系统的相对来说开发难度大些，硬件设计也有所不同。因此在进入带操作系统的产品开发前，首先要以熟悉不带操作系统的产品开发作为过渡。而且，是否选择用操作系统要根据具体情况定，并非带操作系统就一定高级，而是要按照需要来定。

本章讲解采用 ARM 开发一款家用电器：智能室内导航家用拖地机器人。当外出的时候只要按下一个按钮，回家后房间的地面就会被打扫得干干净净。

本章参考的设计产品样品是 mint5200 拖地机，该机是美国设计的，在国内生产，销售在国外。图 6-1 是美国 mint5200 拖地机的外观，顶上的大窗口是红外导航信号的输入窗口。上面的三个按钮分别是开启、干拖、湿拖三个功能。干拖和湿拖的差别是局部行走不同，干拖是直接向前，而湿拖则是左边和右边分别拖。从全局角度分析，导航后的行走路线基本类似。本章主要关注如何实现内部线路和软件设计，细微的差别不做深入介绍。



图 6-1 mint5200 拖地机的外观



图 6-2 拖布固定座

拖地机下方有一个拖布，通过埋在塑料壳内的磁铁吸附在机身上，如图 6-3 和图 6-4 所示为拖布固定座和拖布。



图 6-3 mint5200 的拖布

拖地机通过室内导航仪进行导航，如图 6-4 所示。



图 6-4 mint5200 的北极星导航仪

mint5200 机器人能自主思考，自主移动，自动避开障碍物，沿边清扫，实现室内全覆盖清扫。

在大致介绍完拖地机样品以后，我们来解析拖地机内部，其包括机械机构和电子部分。然后确定设计方案，并考虑如何应对知识产权的问题。

6.1 对 mint5200 进行拆解

图 6-5 是 mint5200 的内部结构,可以看到拖地机只有两个轮子,在轮子后方有铁质平衡块,在平衡块中间是镍氢电池盒,两个轮子通过减速电动机输出。在减速电动机的后端有一个测速片,如图 6-6 所示,测速片边上有一个带孔的测速片,槽形光耦(见图 6-7)安装在测速片上方,每转过一个齿,光线被阻挡一下,信号就发生一次变化。通过电路可以计算出当前的速度,并计算出走过的距离。在拖地机前端有 4 个传感器,分别是两个轻触开关组成的左右碰撞微动开关传感器和两个防止掉落的红外槽形光耦传感器。

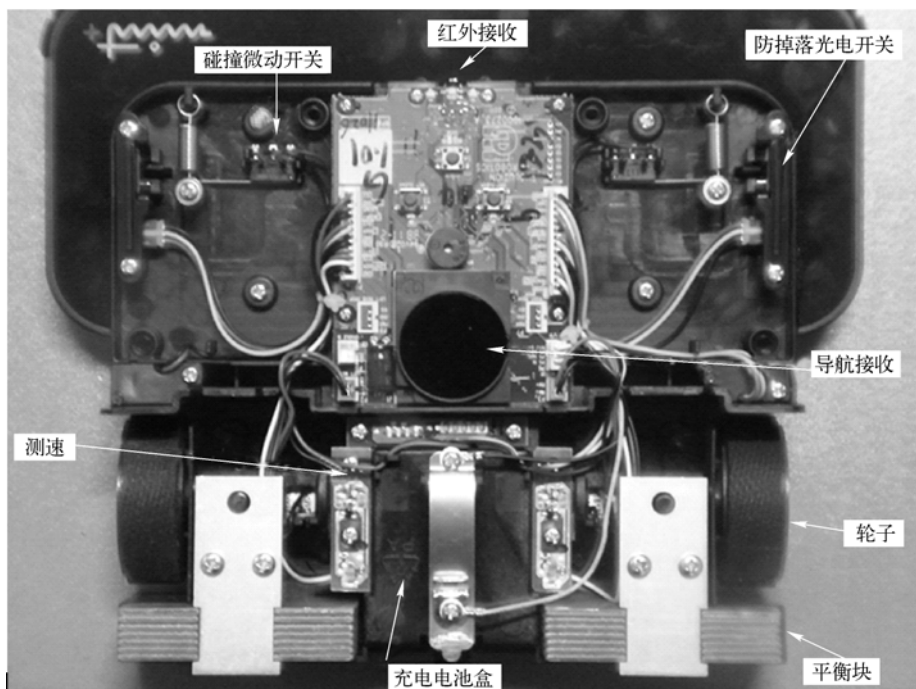


图 6-5 mint5200 的内部结构

下面对电路板进行分析。电路板正面主要是一个方的模块,上面有一个圆形的窗口,打开窗口,可以看到如图 6-8 所示的结构。拆开里面的接收管,发现是普通的红外接收管,在窗口的小电路板背面有一个 freescale 的 DSP MC56F8013 (见图 6-9)。针对其声明的导航系统专利,专利提到这是 PSD (位置敏感器件) 技术,也就是外部红外导航仪发射三束红外光线到屋顶,光线反射回来,在接收端类似 CMOS 摄像头的 PSD 传感器上留下光斑,根据光斑距离并经过几何运算推算出目前和光源之间的距离及方位。但是实际上传感器并不是 PSD 传感器,因此分析其实现技术和专利声明的技术并不一致。此外,针对北极星导航仪(见图 6-4),拆解发现(见图 6-10)红外发射不是三个,而是两个发射管。用示波器检测发现,两个发射管上的波形并非一样,而是有不同的频率;另外,在电路板背面是一个单片机(见图 6-11)。综合分析得出导航方式不是 PSD 方式,因为 PSD 器件很贵,在家用电器上安

装 PSD，销售价格将会非常贵，并不适合。从硬件实现推断，导航采用的是红外的方式测距离，接收端（见图 6-8）有三个接收管，说明接收到的相位有个差，由于 DSP MC56F8013 的存在，可以推断采用的是红外相位测距方式，然后通过计算得出目前的方位和距离。

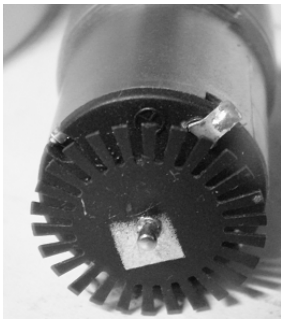


图 6-6 测速片

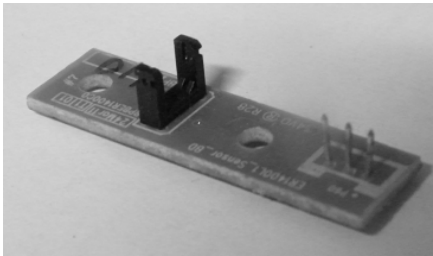


图 6-7 槽形光耦

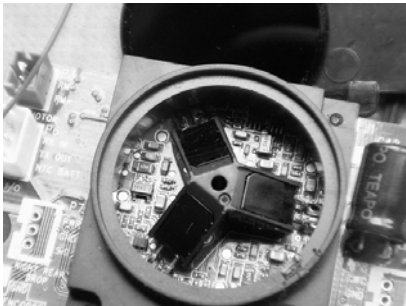


图 6-8 拖地机导航模块接收部分



图 6-9 导航模块背面

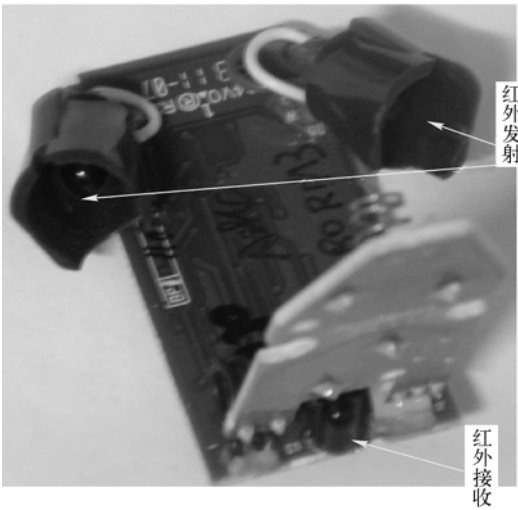


图 6-10 北极星导航仪电路板

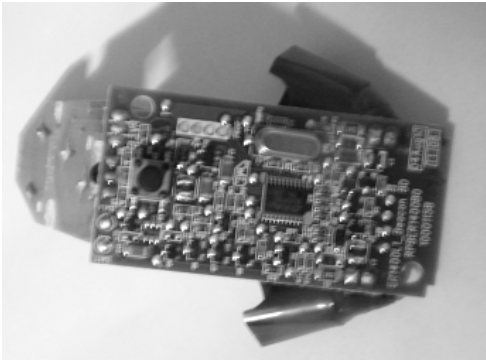


图 6-11 北极星导航仪的单片机

至于主板前端（见图 6-12）的一体化红外接收头和两个红外发射头，红外发射头直接和导航仪通信，红外接收头接收导航仪发来的命令信号。从实现硬件上分析，应该是红外通信和红外导航分开进行的。

观察拖地机的运行，发现主要有三种行走路线，在空旷地带是 Z 字形行走（见图 6-13），在家具脚边是围绕家具脚走，在墙壁边是沿边缘走。

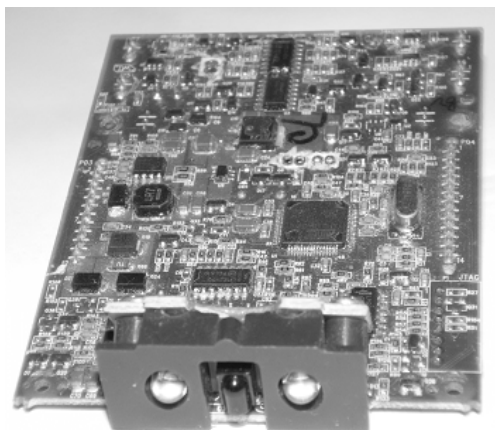


图 6-12 主控板背面

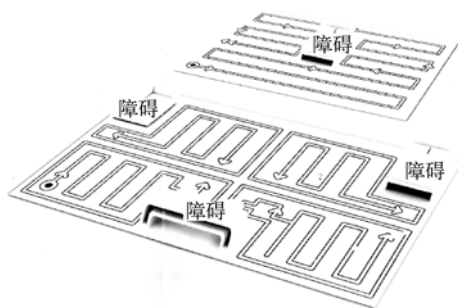


图 6-13 Z 字形行走

由于碰到家具的脚后发现轮子有打滑现象，因此推断用测速计数的方式来记忆行走的路线不现实，一定还有其他途径来解决拖地机侧面撞到家具的脚后位置的变化问题。仔细研究主板，发现有一个 ex3500 单轴模拟陀螺仪，机器一定是通过这个陀螺仪感知侧向的碰撞的，从而探测到家具脚的形状和大小。在单轴陀螺仪的边上还有模拟运放，其作用是将信号放大后送入 CPU AT91SAM7S256AU 的 A/D 转换器。CPU 是 ARM 7 处理器，由于系统的导航采用了 DSP 对红外相位信号进行快速处理，因此 ARM 主要负责传感器的输入处理、异常的判断、陀螺仪和算法，清扫路线覆盖算法等。本章的重点是设计产品的主要部分，因此覆盖算法不进行讲解，设计部分也不会涉及。

6.2 设计方案

经过以上的分析，可以确定的是我们所设计的家用拖地机功能必须和 mint5200 一样，而电路板要自行设计，并且程序也要自行设计。原因是一方面国外有专利技术保护，另一方面主要芯片都是加密的。如果不掌握核心技术，产品将无法升级，企业就谈不上生命力。

除了电路板要自行设计外，外观也必须有所不同，从而避免与 mint5200 的外观专利问题。

6.2.1 外观的修改

参考 mint5200，委托产品设计公司进行外观的设计，一般是首先构思出草图，提供多幅草图供选择。然后选择合适的样图，让设计公司细化，形成三维图，如图 6-14 所示。从

图上可以看到，顶盖和盖子前方与 mint5200 有很大的不同，盖子的高度也发生了变化，符合专利法的相关条款，不会与 mint5200 的外观专利发生冲突。



图 6-14 新设计的三维图

外观设计完成后，要形成六面视图，如图 6-15 所示，然后申请外观专利。

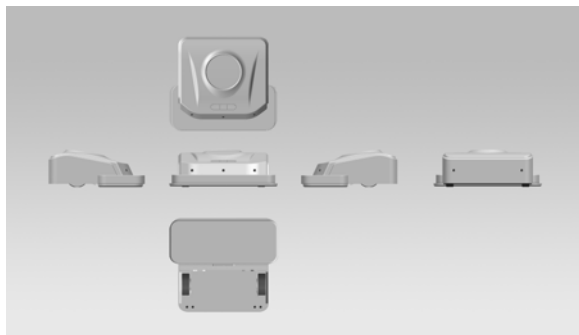


图 6-15 外壳的六面视图

6.2.2 机械设计

外观设计完成后就进入机械设计阶段，机械设计可采用 UG 或者 Pro/E。图 6-16 是部分零件的设计图。

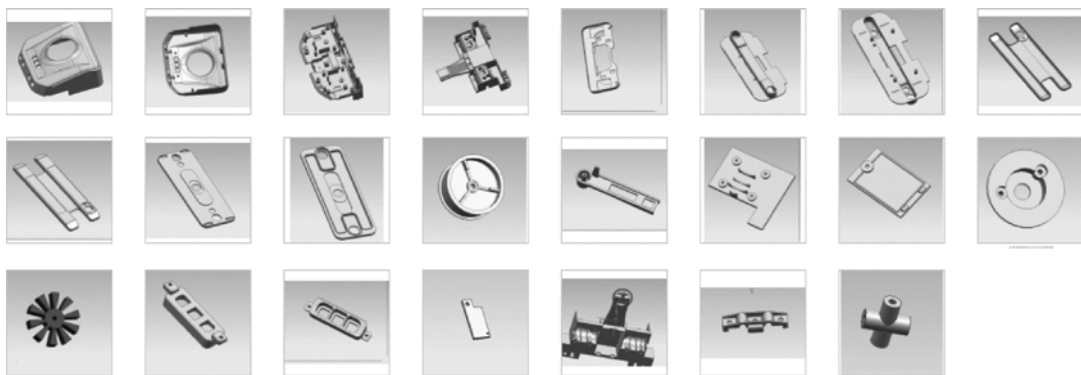


图 6-16 UG 设计的部分零件图

UG 的图纸可以直接拿去做快速成型。如果用 3D 打印机, 价格将非常昂贵。因此若有些零件精度要求不是特别高, 建议在加工中心做。个别要求高的零件, 可以在快速成型机器上做, 这样成本比较节省。塑料样品做好后要装配, 图 6-17 是拖地机装配后的效果, 图 6-18 是导航仪装配后的效果。由于红外透光板无法快速成型, 因此采用普通有机玻璃。只有等模具开出后, 才能用专用的塑料材料加工红外透光板。机械的设计并非一次能完成, 需要配合电子控制部分进行实际工作后经过多次修改才能最后定下来。



图 6-17 拖地机装配后的效果



图 6-18 导航仪装配后的效果

6.2.3 室内导航方案的选择

方案的选择和产品的性能密切相关。不同的要求可以采用不同的方案, 从而产生不同的设计, 最终涉及产品的成本、研发成本、销售的价格定位及市场竞争力。因此产品方案的选择很关键。

为了让产品实用, 需要在短时间内实现室内高效率的打扫, 减少重复清扫, 让有限的电池能量尽可能多地打扫房间区域。因此不能采用随机算法, 如果用随机算法, 整个系统就非常简单, 只要用一个单片机, 两个碰撞传感器即可, 一旦碰到障碍物就随机掉头。但问题是有很多区域没有打扫到, 有些区域却重复打扫。因此需要有路线的规划。

拖地机器人最关键的是全覆盖清扫算法, 而算法的技术是下层软件正常工作, 硬件稳定。例如, 一般的算法不允许轮子打滑, 只要轮子打滑就不能让算法正常。而在实际使用中, 轮子不可能整个过程不打滑, 所以这些算法只是理想条件下的研究。以 mint5200 拖地机样品为例, 假设左、右两个轮子的直径为 45mm, 只要打滑 1cm, 方向就偏移了二十多度。而正常运行时, 电动机减速齿轮、轮子的大小不一致都会引起误差的积累, 最终左、右两个轮子走过的路线长度是不同的, 也就是说拖地机无法保证直线行走。因此一定要用到外部的修正技术, 确保拖地机至少是局部的直线行走, 只有直行线走能有保证, 覆盖算法才能正常工作。当然覆盖算法很复杂, 除了直线行走, 还要处理沿边行走。国外的研究资料表明, 有导航和无导航的家用保洁机器人售价呈 3 倍的差距。

我们分析一下常见的室内保洁机器人的定位技术。采用输入房间几何地图的方式并不现实, 因为桌子、椅子等可能随时移动。如果采用多个外部噪声波发射器, 由拖地机的噪声波接收器接收测距定位, 带来的问题是多个外部发射装置不方便家用安装, 而且噪声波反射干扰等也影响定位。现在也有不少智能扫地机采用摄像头定位, 但是受光线干扰大, 位置准确度一般。红外线相位测距技术也可以用来定位, 但实验发现日光灯干扰明显, 因此并不稳定。如果采用 PSD 器件定位, 则定位效果不错, 但二维 PSD 传感器的价格要几千元, 并不

适合在家电中使用。如果采用目前扫地机器人常用的虚拟墙或者灯塔技术，则只能大致判断范围，并不适合室内导航。各种设计方案的比较见表 6-1。

表 6-1 设计方案的比较

设计 方 案	优 点	缺 点
随机路线	实现简单，可靠	有很多区域没有打扫到，有些区域却重复打扫
多个外部噪声波辅助定位	实现简单	安装不方便，噪声波反射干扰定位的准确性
摄像头定位技术	性能尚可	受光线干扰大，位置准确度一般
红外线相位测距技术	比较精确	系统相对复杂，受日光灯干扰明显
采用 PSD 器件定位	精度高	传感器价格昂贵，不适合在家电上应用
虚拟墙或者灯塔	简单有效	只能大致判断范围，不适合导航

以上技术各有优缺点，设计时要尽量考虑融合不同技术的优点。分析表 6-1，结合我们要设计的产品，选择红外的方案。用红外和噪声波结合的方式来实现，这样做成本比较低，而且稳定性相对较高。当然，这里面涉及一些特殊的方法，将在后面讲述。

6.2.4 导航系统方案设计

从前面的介绍中可以发现，拖地机的特点是有两个后轮，分别用电动机驱动。虽然用 PID 调节方式控制两个轮子使其速度一致，但由于打滑和机械误差，拖地机每运行一段距离就会发生偏移，也就是说在实际运行的过程中无法走直线。这样产生的问题是当多个回合走下来以后，拖地机将产生较大的偏转。但是拖地的路径覆盖算法要求必须走直线，否则将无法正常工作。为了克服这个困难，一种方法是在拖地机内部安装电子罗盘 HMC5883L，但电子罗盘容易受磁场干扰，特别是在拖地机下方存在强磁铁的情况下，更加难以实现；另一种方法是在拖地机内部安装 MEMS 陀螺仪，但是陀螺仪有漂移和累计误差，以及 A/D 转换器等引起的误差，时间一长也会发生偏转。为此，需要提出一种定期纠正角度和位置的方法。

因此，本产品采用 6 轴内部陀螺仪导航和外部红外导航相结合的技术，通过卡尔曼滤波和三维旋转矩阵运算获得稳定的角度偏移来实现局部导航，通过外部的红外修正实现全局的位置修正。针对轮子打滑无法避免的现象，结合硬件设计中的陀螺仪和加速度传感器，动态实时检测姿态异常，通过算法调整路线的规划，实现沿边缘清扫和室内全覆盖清扫。

由图 6-19 可以看到，导航系统包括外部导航仪和拖地机上的陀螺仪，导航仪和拖地机均放在需要打扫房间的水平地面上，导航仪放在需要打扫房间的某一边的中间。

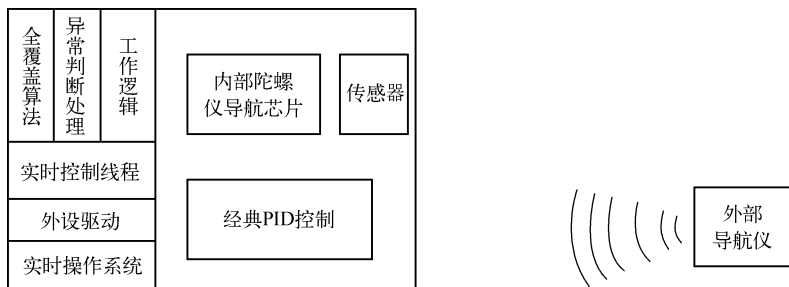


图 6-19 拖地机的两种导航结合

拖地机有两个后轮，分别用电动机驱动，虽然采用PID调节方式控制使两个轮子的速度一致，但由于打滑和机械误差，拖地机每运行一段距离就会发生偏移，即无法走直线。为此，这里提出一种定期纠正角度和位置的方法，如图6-20所示。

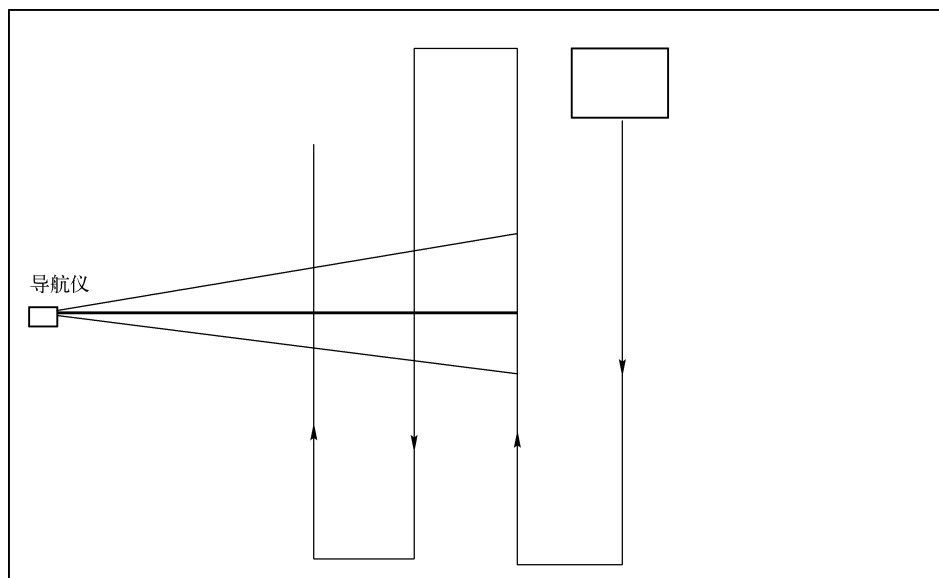


图 6-20 外部红外定期角度修正

导航仪放在房间边上的中部，发出红外线和噪声波。拖地机的运行角度始终与导航仪垂直。拖地机从上往下、从右向左运行，这样每次经过房间中央，拖地机都会获得红外和噪声波信号，从而知道自己目前的位置。即使走偏了，也能及时纠正。

内部的陀螺仪还是有用的，因为在有些位置，特别是角落，由于物体的阻挡，无法接收到导航信息，这样陀螺仪便能协助拖地机获得方位信息。采用数字式高精度陀螺仪并结合卡尔曼滤波算法，可以保证每分钟误差在 1° 以内。这样，在角落工作 10min 引起的误差是 10° 左右。

导航工作的原理为，如图 6-21 所示，导航仪发出基本无散射的红外线（如图中粗线所示），同时也发出噪声波，噪声波具有一定的角度（图中是扇形的两条边）。红外线的作用是作为参考基准线，噪声波的作用是测距离。通过参考线和距离，可以测定方位。测定方位的原理为：当拖地机找到参考线后，首先测到与导航仪之间的距离为 L_1 ，这也是圆半径 r ；然后向前走距离 L_2 ；之后测量到与导航仪的距离为 L_3 ，假设控制 L_2 为很小的距离，则 L_1 和 L_3 之间的夹角将很小，从几何关系图上可以知道， L_1 和 L_3 之间的扇形圆弧变化也很小，可以用直线 L_4 代替圆弧。由于 L_1 和 L_3 的夹角很小，而 L_1 和 L_3 很长，因此可以认为 L_4 和 L_1 及 L_3 的夹角接近 90° 。

假设 $L_5 = L_3 - L_1$ 则 L_2 、 L_4 、 L_5 构成直角三角形。通过求 L_2 和 L_4 之间的反正弦函数可以获得 L_2 和 L_4 的夹角 α 。然后 $\alpha + 90^\circ$ 便是 L_1 和 L_2 之间的夹角。从而可以知道目前扫地机的水平姿态角。

具体实现方法如图 6-22 所示。

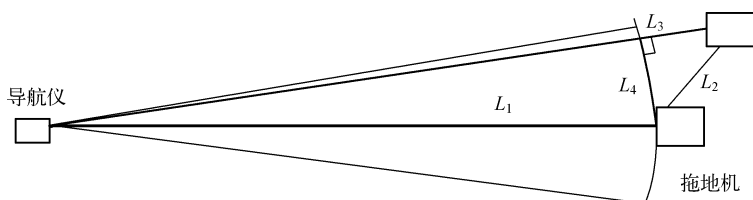


图 6-21 拖地机位置的几何关系

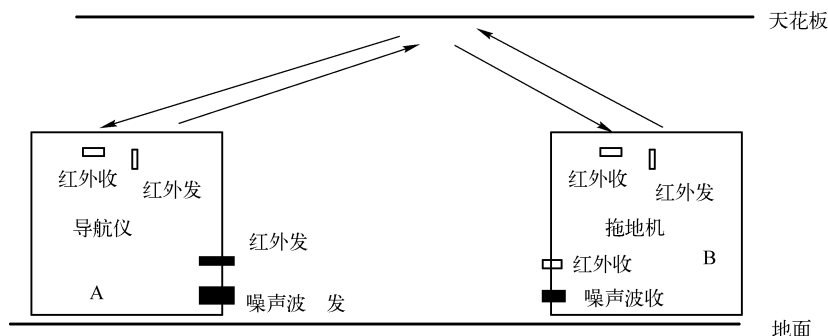


图 6-22 导航仪和拖地机之间的通信

导航仪和拖地机放在水平地面，在导航仪上有 3 个红外发光管，分别射向正上方、正前方、左边和右边。其中正上方的发光管为 120° 发射角度；前方的红外发射管通过一个长的圆孔，发射角度为 1° ；左边和右边则采用 30° 的发射。但左右两边的红外发射功率很小，调整有效发射距离为 10cm 左右。导航仪上还安装了一体化的红外接收头，为 120° 接收角度，正面朝上。另外，在导航仪上也安装了噪声波发射头。

拖地机上安装有一个红外发射管，射向正上方，为 120° 发射角度；还有两个一体化的红外接收头，接收头正面朝向前方。

工作过程为：拖地机在运行的过程中，每过一段距离，通过上方的红外发射管，发射测距请求数据包 CQ1，物理通信方式为 38kHz 的调制红外线。红外线从天花板反射到导航仪，导航仪上方的高增益一体化红外接收管经过解调并处理，输出解码的波形。导航仪上的单片机分析波形，知道被请求测距。导航仪通过前方的噪声波发射头发出噪声波，同时通过前方的红外发射管发射红外线。由于这个红外线发射角度为 1° ，因此基本上没有发生散射，接近直线，同时也具有一定的范围。也就是说光柱是一个以 1° 为夹角的锥形光柱。这样可以使得拖地机的红外接收管容易对准并接收到。如果采用激光，反而容易引起收发管之间对准精度要求过高的问题。红外接收管接收到以后，就获得了参考线 L_1 。噪声波发射头发射噪声波后，经过若干时间，到达拖地机的噪声波接收管，拖地机内的处理器分析计算出距离 r 。然后拖地机继续前进很小的距离 L_2 ，同时检测到和导航仪之间的距离 L_3 ，按照前面提到的检测原理，可以获得目前的水平姿态角，从而可以知道目前在房间内的位置信息。

导航仪左右两边安装的红外发射管是为了防止拖地机运行时撞上导航仪，从而导致导航仪产生位移，影响位置基准。当拖地机从左边或者右边接近导航仪时，通过控制导航仪发射红外的强度，可以在一定距离内探测到红外信号，从而可以避开导航仪。

导航仪的另外一个功能是多个房间之间的导航。多个房间分别安装导航仪的情况如图 6-23 所示，每个房间各自有不同的地址编码，拖地机接收到多个应答信息后，可以知道有多个房间需要打扫，从而为进入相应的房间做准备。假设大厅旁边有两个小房间，可以通过房门进入。三个房间内都安放导航仪。

导航仪上方有红外发射管和接收管，可以按照拖地机主机的要求发出导航仪的 ID 号，拖地机主机接收到以后即可知道目前所处的房间。

拖地机主机上装有功率较大的红外发射管，向 3 个方向发射，成 120° 角排列。假设拖地机经过 1#房间和 2#房间的交界处，2#房间的导航仪接收到红外信号，则也发射红外信号，其数据段包括导航仪本身的 ID 号，拖地机记录下目前的位置，以便将 1#房间打扫结束后再打扫 2#房间。如果拖地机进入 2#房间，由于 1#房间的导航仪不再感应到拖地机发出的红外信号，因此导航仪自动休眠。

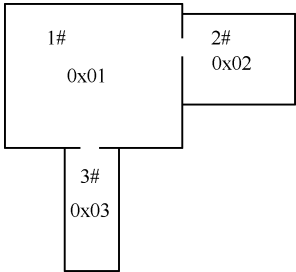


图 6-23 多个房间的处理

邻近房间的判断：当拖地机在清扫房间的同时到达两个房间交界处时，如 1#房间和 2#房间交界处，拖地机的红外接收管能同时接收到两个房间的 ID 号。因此，拖地机可以知道 1#房间和 2#房间相邻，并且按照路线可以知道邻近房间的入口位置。当主机判断目前 2#房间清扫完毕后，便返回 1#房间和 2#房间交界处，由于 1#房间的导航仪接收到拖地机的红外信号，因此导航仪开始发射红外信号。回到 1#房间后，按照同样的工作原理，拖地机可以进入 3#房间。

如上面三个房间，当 1#房间打扫结束后，拖地机控制主机将 1#房间从打扫列表中去除，然后通知 2#房间导航仪开启，同时按照刚才记忆的路线走到 2#房间入口区域，2#房间导航仪开始发射红外信号，拖地机进入房间开始清扫。

所有导航仪都是间歇工作模式，只有当接收到拖地机的红外信号后才进入工作状态，每过若干秒进入休眠状态。

导航仪和拖地机之间需要双向通信，包括两种类型：

种类 A	信息方向
基准请求	拖地机到导航仪
基准应答	导航仪到拖地机

种类 A 为立刻发生，是为了测量获得距离。拖地机发出请求信息后，便立即开始接收导航仪发出的噪声波，通过时间差得到距离。

种类 B	
导航仪身份请求	拖地机到导航仪
导航仪应答	导航仪到拖地机

种类 B 为导航仪延迟一段随机时间应答。由于多个房间可能有导航仪，拖地机要清扫多个房间时，在交界处可同时收到两个应答信号，为了防止接收冲突，采用延迟随机时间的方式。

6.2.5 红外通信方案设计

拖地机的中央处理器要处理很多事务，采用软件解码的方式则红外要占用比较多的处理器时间，而且要求对信号处理的优先级很高。而在运动控制应用场合如果红外处理的时间过于频繁、占用的时间过长，必将影响系统的动态响应性能。处理器还要控制电动机，执行 PID 速度控制、陀螺仪的解算。因此红外收发最好用硬件来实现。采用硬件实现红外的遥控信息处理，将大大减轻处理器的负担。但是通用的红外编解码硬件芯片通信格式固定，不能满足自定义的格式要求。其次，在不少系统中已经拥有 CPLD，如果再加红外芯片，必将增加成本。因此，用 CPLD 中的一部分来实现红外通信是一种不错的选择。

红外遥控分为发送和接收两部分。发送的信息通过 38kHz 调制后经红外发射头发射出去。因此，这里需要解决的是如何将信息进行编码并调制。

红外通信格式通常如图 6-24 所示，首先是大约 9ms 的低电平，然后是 4.5ms 的高电平，之后是数据，这里定义为 16b 的数据，通信结束后信号线为高。



图 6-24 红外通信格式

中间的数据通信格式如下。

对于 0，定义为前面 2/3 是低电平，后面 1/3 是高电平，如图 6-25 所示。

对于 1，定义为前面 1/3 是低电平，后面 2/3 是高电平，如图 6-26 所示。



图 6-25 0 的数据通信格式

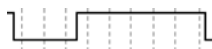


图 6-26 1 的数据通信格式

因此需要将每一比特对应的时间分为 3 份，在每一份时间的开始计算目前需要发高电平还是低电平。定义每一比特的时间占 1.5ms，分为 3 份则每份占 0.5ms，这样编码发生的时钟周期为 0.5ms。

最后这些编码经 38kHz 调制后输出。

假设 CPLD 的输入时钟是 8MHz，首先需要分频为 38kHz，采用 VHDL 编写代码进行分频。对应的 VHDL 代码如下，其中 en2 是使能信号。

```
infrad: process (clk,en2)
    variable index:integer range 0 to 300;
begin
    if(clk'event and clk='1') then
        if(index>=211) then index:=0;
        else index:=index+1;
        end if;
    if(en2='1') then
        if(index<100) then infrad38K<='1';
```

```

else infred38K<='0';
end if;
else infred38K<='0';
end if;

end if;
end process;

```

分频后的信号一方面作为调制信号，另一方面再次分频，作为编码发生的基准时钟。分频原理同上，分频后的输出是占空比为 50%、周期为 0.5ms 的时钟。

发送的总体硬件如图 6-27 所示，首先是 8MHz 的时钟经过 S38K 分频模块，输出为 38kHz 的时钟，该时钟一方面接到输出与门对编码进行调制，另一方面输出到 fenpin 模块，输出 0.5ms 的时基信号。此信号进入 serialcodegenerate 串行编码发生模块，该模块负责产生编码。假设应用只需要产生两种通信数据，因此 modesel 负责选择目前采用哪种。当然，要产生多种其他的代码，只要适当修改 VHDL 的编码即可。

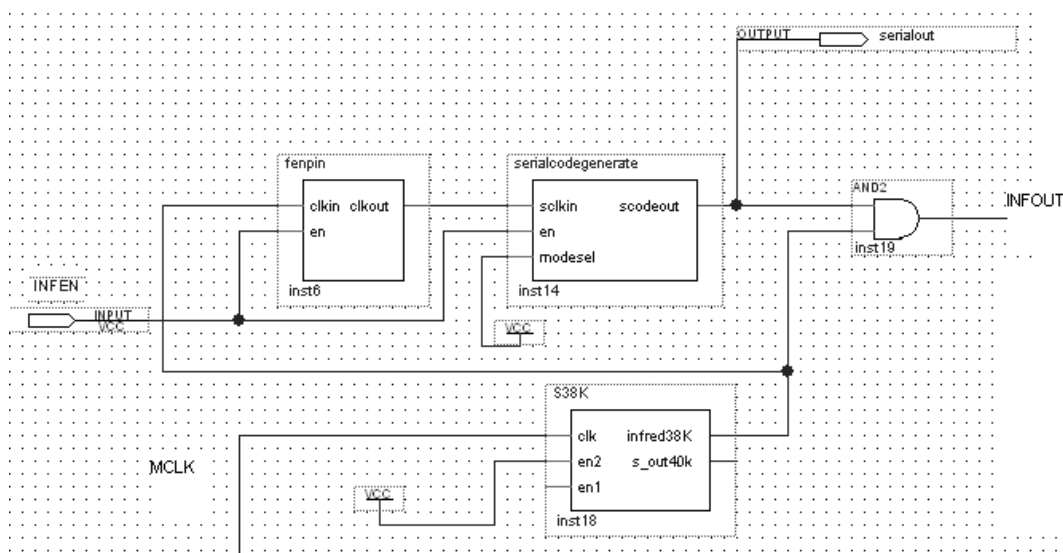


图 6-27 红外发送的 VHDL 硬件原理图

编码发生模块首先需要产生 9ms 的低电平和 4.5ms 的高电平，然后是 16b 的数据。由于每比特对应 3 个时钟，因此需要定义一个内循环变量，每满 3 次就恢复为 0。然后是定义外循环变量，9ms 的低电平相当于 18 个 0.5ms，4.5ms 高电平相当于 9 个 0.5ms。发送比特的时候，不管是 0 还是 1，前面 1/3 的时间一定是 0，最后 1/3 的时间一定是 1，只有中间 1/3 的时间对应 0 或者 1 变为低或者高。因此，当内循环变量为 1 时才判断输出是 0 还是 1。

图 6-28 是仿真结果，MCLK 是 8MHz 的主时钟，INFOUT1 是输出到发射管的信号，INFEN 是使能。serialout 是编码输出，发射的代码是 0x5151。因此，每次只要让 INFEN 变低然后置高，就能产生遥控发射波形。

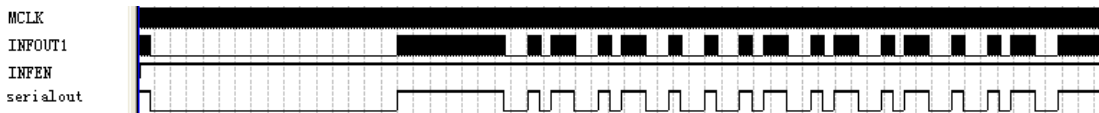


图 6-28 红外发射的仿真结果

下面介绍红外接收。

在拖地机主控板上，如果用 CPU 实现红外接收，将占用相当多的时间。红外接收对实时的要求很高，为了信号处理的实时性必须用中断来实现红外接收的解码，但这样又降低了对其他中断的响应实时性。而用 CPLD 设计的红外解码将由硬件自动解码，等接收成功后向 CPU 申请中断，CPU 读取接收到的红外遥控数据。

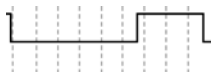


图 6-29 接收的 1b

对于接收到的每一比特，红外接收首先要分析接收到的是 0 还是 1。和发射的时候一样，将每一比特分为 3 格，通过分析每格内的电平来确定比特，如图 6-29 所示。例如，第一次是 0，第二次是 0，第三次是 1，对应的比特是 0。

在每格内，考虑到接收的稳定性和可靠性，将每个时间格细分为 n ，在 n 的中央检测电平，确定是高还是低。假设采用 38kHz 的输入时钟，时钟周期为 $26.3\mu\text{s}$ ，则 $n=0.5\text{ms}/26.3\mu\text{s}$ ， $n=19$ ，在 $n=9$ 的地方也就是接近中央处检测电平。为了节约资源，在 VHDL 中用整数表示变量。

首先假设用 38kHz 作为基准，并在红外起始的地方开始检测，一直检测到通信结束。

38kHz 的时钟周期是 $26\mu\text{s}$ ，每个时钟误差为 $0.3\mu\text{s}$ 。如果发送 16b，加上起始电平，需要 75 个 0.5ms，每个 0.5ms 需要 19 个 38kHz 的时钟，因此一共需要 1425 个 38kHz 时钟脉冲。到最后一比特时累计误差是 $1425 \times 0.3 = 427.5\mu\text{s}$ ，但正常工作时，在 0.5ms 的中间采样也就是 $250\mu\text{s}$ 的地方，这里的误差远超过可承受的范围，接收将无法工作。因此，考虑在每一比特的地方纠正，同时针对起始的低电平和高电平分别进行采样时间纠正。

在导航仪上，由于任务不多，可直接用单片机解码红外接收，红外发射也直接用单片机实现，不必另外增加硬件，这样可以大大节约成本。

6.2.6 保证直线行走的设计方案

拖地机器人最关键的是高效率的全覆盖算法，算法稳定工作的前提是轮子走的路线要能确切知道。例如，在轮子打滑的时候，由于机器本身无法得知发生了打滑，因此算法还是依照航向改变以前的参数运行，最终导致错误。除去轮子打滑的因素，大部分时间拖地机运行也需要维持直线方式的行走。因此两个轮子需要进行速度 PID 控制，尽量让两个轮子走过的距离大致相等。当然在经过一段距离后，并不能确保方位没有发生偏离，实际试验结果证明也的确无法保证，因此还需要在内部加陀螺仪予以实现。这里先研究轮子控制的问题。

电动机采用 250r/min 的直流减速电动机，电动机后面安装测速转盘，有 26 个孔，如图 6-6 所示，用槽形光电传感器检测。最高速度设置电动机减速输出每秒 4 转，按照减速电动机的参数，减速比为 26:1，因此实际的电动机转速是 $4 \times 26 = 104$ 转/秒，测速盘输出的脉冲则是 $104 \times 26 = 2704$ 个脉冲/秒。

测速脉冲 $N=676 \times n$ ，其中 n 是减速电动机输出轴每秒的转速。

定义：快速为轮子每秒 4 转，中速为每秒 2 转，慢速为每秒 1 转。

转换为脉冲是：快速 2 704 个每秒，中速 1 352 个每秒，慢速 676 个每秒。

分为 3 档：慢速情况将 PWM 脉宽加宽或者减小 5；

中速情况将 PWM 脉宽加宽或者减小 10；

快速情况将 PWM 脉宽加宽或者减小 12。

电动机输出轴转 1 圈，测速盘转 26 转，脉冲数为 676 个。

轮子一圈的外周长是 13cm，因此 676 个测速脉冲对应行走 13cm，故 1cm 对应 52 个脉冲。

有关两个轮子的速度均衡问题：由于两个轮子的电动机特性不完全一致，同样的 PWM 占空比会导致电动机速度有微小的差异，误差的积累可能使拖地机实际的行走路线不是直线，而是偏向一边。为此，需要对两边的电动机速度进行协调，使电动机转速一致。采用经典自动控制中的 PID 调节可以实现。

6.2.7 方案的合理性分析

按照以上大致的方案设计的机器人能自主思考、自主移动、自动避开障碍物、沿边清扫、进行室内全覆盖清扫。为了让产品实用，需要在短时间内实现室内高效率的打扫，减少重复清扫，让有限的电池能量尽可能多地打扫房间区域，为此，产品采用 6 轴内部陀螺仪导航和外部红外导航相结合的技术，通过卡尔曼滤波和三维旋转矩阵运算获得稳定的角度偏移来实现局部导航，通过外部的红外修正实现全局的位置修正。针对轮子打滑无法避免的现象，结合硬件设计中的陀螺仪和加速度传感器，动态实时检测姿态异常，通过算法调整路线的规划，实现沿边清扫和室内全覆盖清扫。将内部陀螺仪导航和加速度传感器融合，解决了传统清扫算法不允许轮子打滑引起位置偏移、清扫算法不实用的问题。通过外部红外基准和超声波相结合，同时结合空间位置算法，解决了室内全局位置估算、多房间打扫的问题。

产品要求具有以下的技术特色。

- 陀螺仪局部导航：单轴陀螺仪无法检测出机器人的三维姿态，毕竟拖地机运行时不可能绝对水平。因此必须用三轴的陀螺仪实现，然而三轴陀螺仪需要进行旋转矩阵运算，一般必须采用 DSP 完成，导致成本很高。为了解决这个问题，本项目采用 ARM cortex 技术和 CPLD 相结合，采用美国 MPU 6050 高精度陀螺仪，使 ARM 上能运行简化的旋转矩阵，实时给出角度偏移数据。局部导航的应用之一便是角落清扫能力高达 90%，能检测到各种侧向的阻力信号（见图 6-30），可实现复杂地带的规划清扫（见图 6-31）。
- 外部全局导航：机器人通过室内导航技术，杜绝了随机清扫带来的弊病。拖地机器人主机主要完成拖地所需的基本的控制功能，包括电动机速度输出和检测、力矩的检测、传感器输入的处理和人机交互界面处理。拖地机器人主机在完成基本功能的同时，内带陀螺仪导航，完成局部导航功能及轮子打滑、障碍物碰撞检测功能。同时，通过和导航仪的通信，计算出目前的空间位置，实现精确的全覆盖算法。导航仪执行复杂的清扫算法，并将规划好的路线下载到拖地机，从而使拖地机按照导航

仪的命令执行具体工作。在黑暗环境中，视觉识别效果变得很差（见图 6-32），而本系统能正常工作（见图 6-33）。

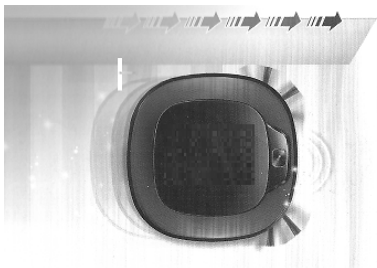


图 6-30 局部角落打扫功能

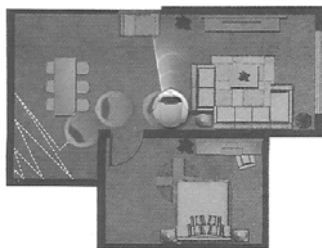


图 6-31 复杂地带规划清扫

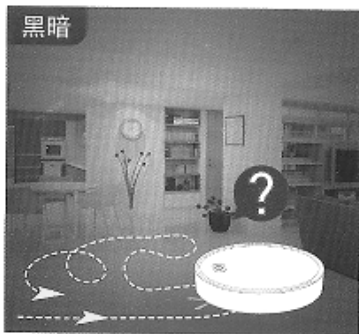


图 6-32 视觉识别定位



图 6-33 外部导航定位

- 改进的算法：通过局部导航检测障碍物和轮子打滑结合全局的外部红外导航，配合上层的全局覆盖算法，本拖地机器人能够实现沿边清扫，路线不重复，清扫效率高、时间短，清扫面积覆盖率达到 95%。效果对比请看图 6-34 和图 6-35。

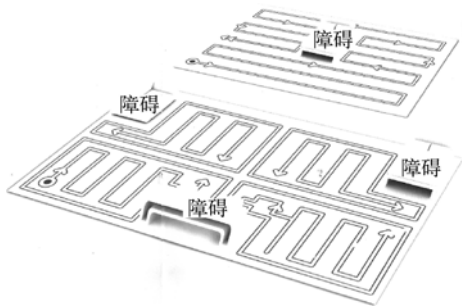


图 6-34 本项目分区域的 Z 字清扫

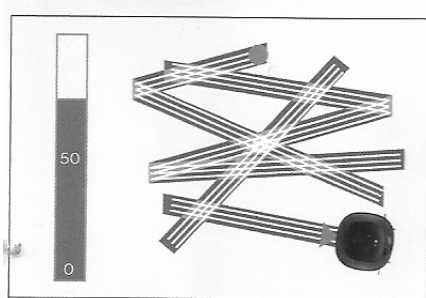


图 6-35 传统的清扫模式

- 学习功能：能够记住障碍物的位置，最大限度避免碰撞。通过记忆，使碰撞次数下降 60%，有效保护了家具，如图 6-36 所示。

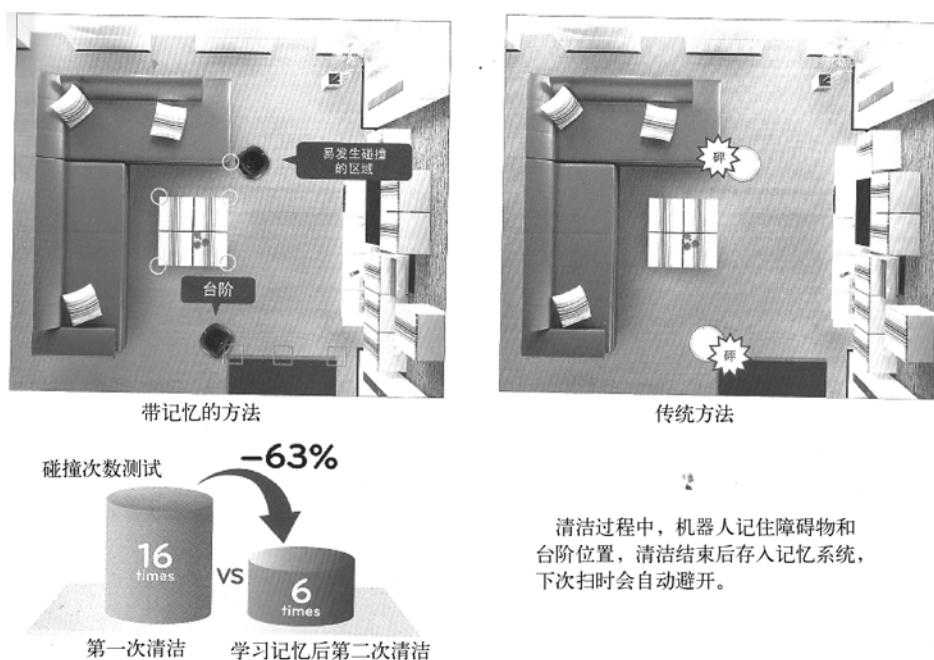


图 6-36 记住障碍物的位置

6.3 总体设计

经过以上的分析，可以知道拖地机内部的大致功能模块，如图 6-37 所示。



图 6-37 拖地机功能分解

设计时要考虑两个方面，首先是拖地机要完成基本功能，其次在基本功能之上要完成全覆盖算法。而基本功能包括了电动机控制和反馈、各种异常的控制，包括电动机过流、电动机空转、掉落检测、遇到障碍物的检测，以及侧向转动时遇到障碍物的检测，等等。另外，还有红外通信系统的设计、方位修正系统的设计、电池电压降低后的检测。除了基本功能以外，还需要软件的算法支持，包括局部的 Z 字形走法、沿边走算法，以及已经走过的路线的记忆，等等。本章重点讲解底层硬件和软件的设计，全覆盖清洁算法并非本书的涉及范围，为了节省篇幅，不再具体介绍。

6.4 硬件设计

按照总体设计思路,首先要设计的是硬件,硬件分为三张图纸,分别是电源部分、CPU部分和 CPLD 部分。首先介绍电源部分。其中电源充电部分,如果把充电管理设置在本电路板上,也是不错的选择,但问题是成本和复杂性将上升。假如将镍氢电池更换为锂电池,或者其他类型的电池,则充电线路要重新改动。为了灵活起见,将充电部分去除,购买外置的镍氢充电器,这样可以把充电部分的设计生产工作规避掉。目前专用的充电器价格比较便宜,质量也不错,总体成本比自行设计进电路板要便宜。

在图 6-38 中, F1 是熔断器,用来防止因故障导致的充电电流异常。重点要介绍的是 S1 和 D1、Q5、Q4,当电池上电后,由于 Q5 的输入为低, Q4 不导通,拖地机电源被切断。当 S1 按下后电流从 S1 到 D1 再到 R19,最后进入 Q5 的基极, Q5 导通,导致 Q4 导通,拖地机供电正常。软件开始运行,在软件运行之初, CPU 置高 DRV,从而有电流进入 R21,进入 Q5,维持 Q4 导通。当电压按钮 S1 松开后,不会导致 Q4 断开。若软件没有将 DRV 置高,则会造成 S1 按下有电、S1 松开电源也断开。R16 和 R18 是对电池进行电压检测,由于电池是 8.4V 的,因此需要分压,然后进入 CPU 的 A/D 转换器。当软件检测到电压偏低时,为防止损坏电池,应该及时自动关闭机器。由于 CPU 和其他器件都是 3.3V 的,因此用一个稳压片 IC2。

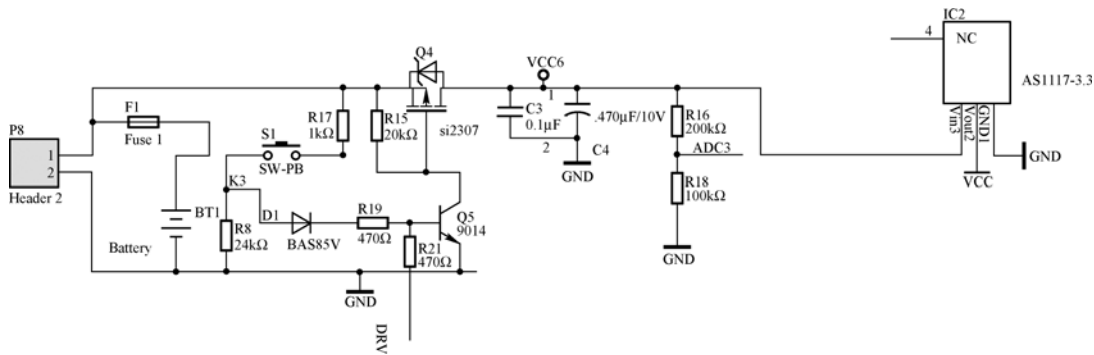


图 6-38 电源部分

图 6-39 是输入/输出部分,包括红外输出传感器 D2、D3、D4,一体化的红外输入头 VS1838B、U5、U6。一个用于红外通信,一个用于红外位置修正。噪声波传感器 P3、P5 用来检测和障碍物之间的距离。轻触开关 P4 和 P7 用来检测是否发生碰撞。P4 和 P7 上分别接 0.01μF 的电容,作用是为了防止干扰,因为电动机运行的时候还是有干扰的。CPLD 采用 EMP240GT100C5,是由于若许多信号都通过 CPU 处理会占用很多宝贵的处理时间,所以用 CPLD 来处理,也就是通过硬件直接对信号进行处理,这样可以大大减轻 CPU 的负担,能确保其实时性能。如果不用 CPLD 而用外扩的数字电路也能实现这一效果,只是器件会比较多。用 CPLD 的优点是有利于设计的加密保护,缺点是生产前必须烧录 CPLD 芯片。

图 6-40 是 CPU 部分,包括两路 H 桥、电动机电流异常检查、LED 和按键、陀螺仪、





JTAG 调试口。JTAG 调试口只在开发的时候使用，出厂时并不焊接。还有速度反馈接口 P9、P10，连接的是速度检测电路板过来的外部脉冲输入信号。CPU 采用 STM32，具有强大的处理能力。上面还有一个 6 轴的数字陀螺仪，具有加速度传感器和陀螺仪传感器，型号是 MPU 6050，是美国 Invensense 公司的产品，具有 16 位精度。陀螺仪的软件程序比较复杂。

图 6-41 是速度检测部分，槽形光耦输入的信号经过 Q1 整形后形成高/低电平，供 CPU 计数。

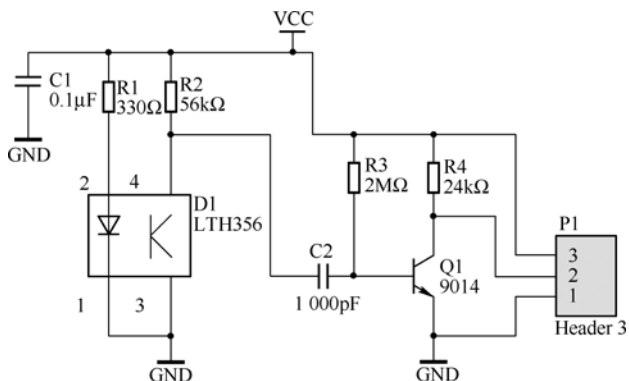


图 6-41 速度检测部分

图 6-42 是噪声波测距电路，噪声波发射部分采用 MAX3232 将电平幅度变大后输出给噪声波发射头，使有效检测距离比较远。如果直接用 TTL 电平驱动，则检测的距离很近，因为没有足够的能量反射回噪声波接收器。噪声波接收部分前面三个放大器是三级放大，最后一个放大器是电压比较器，输出是经过整形的方波，有利于 CPU 检测。运放的 3、5、9 引脚是同相端，电压是 $VCC/2$ ，这样的优点是可以用单电压供电实现放大，并且线性放大的范围较大。噪声波测距电路通过端子 P1 连接到控制主板，也就是连接到图 6-39 中的 P3。共两个噪声波测距电路板，分别探测拖地机左、右两边的障碍物。

上面介绍了拖地机主机的电路，接下来介绍室内导航仪硬件电路。室内导航仪要和拖地机进行红外通信，另外还要检测和拖地机之间的距离，因此要用到一个红外基准发射，还有噪声波发射。具体工作过程请见前面的导航设计部分，这里不再重复。图 6-43 是导航仪的设计，其中 D4 是红外基准线的发射，D4 套在一个铁制的孔里，从而控制红外发射的角度范围在 1° 左右。D1、D2、D3 是红外发射用来和拖地机通信的，它们射到天花板并被反射回来、被拖地机接收到。P2 接 VS1838B，用来接收拖地机发射的红外信号。D5、D7 是防撞用的红外发射管，防止拖地机行走时撞上导航仪本身，从而发生移位。整个导航仪通过一个 MCS51 单片机控制运行。导航仪上的噪声波发射电路和图 6-42 所示噪声波测距电路的发射部分原理一样。

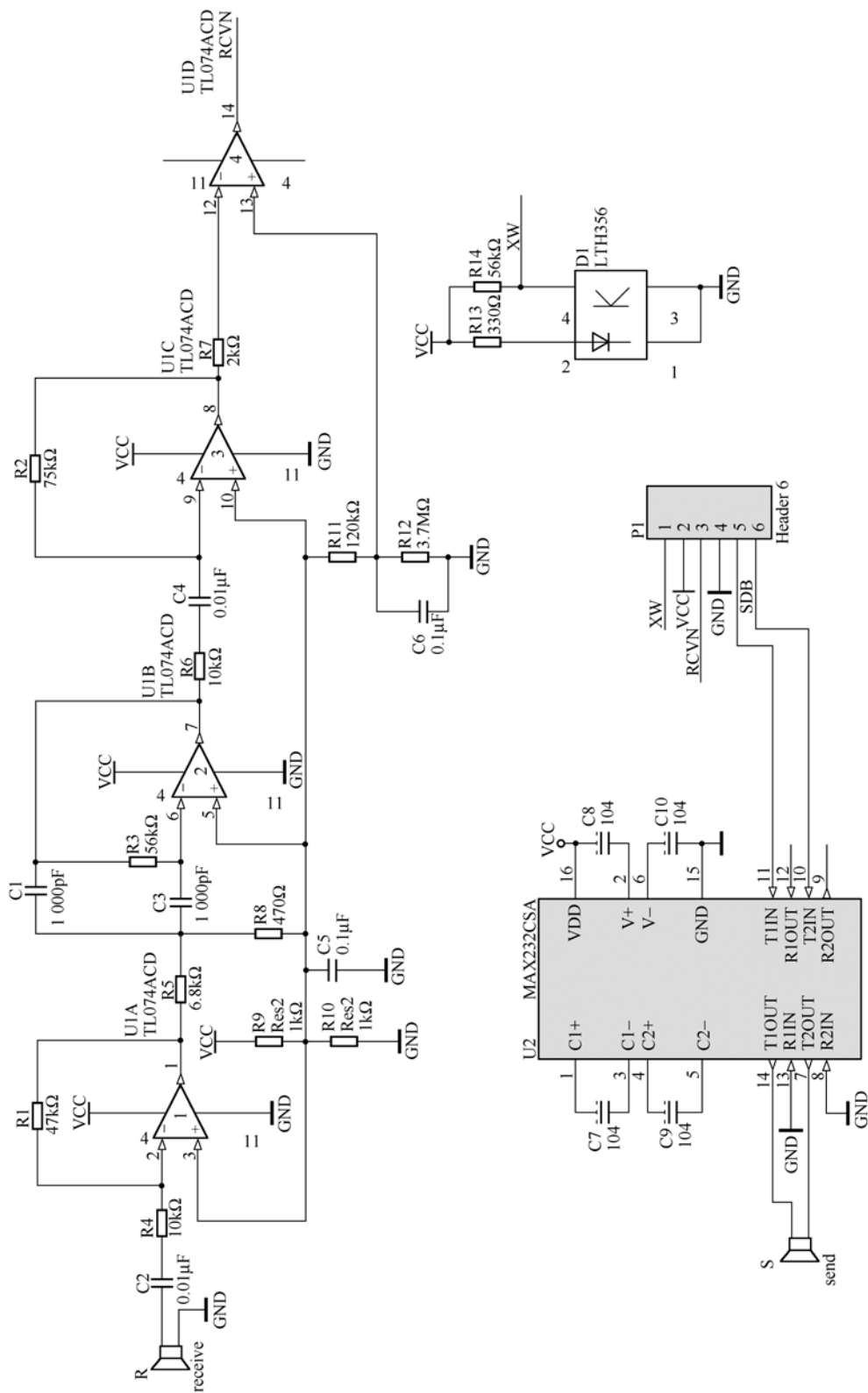


图 6-42 噪声波测距电路

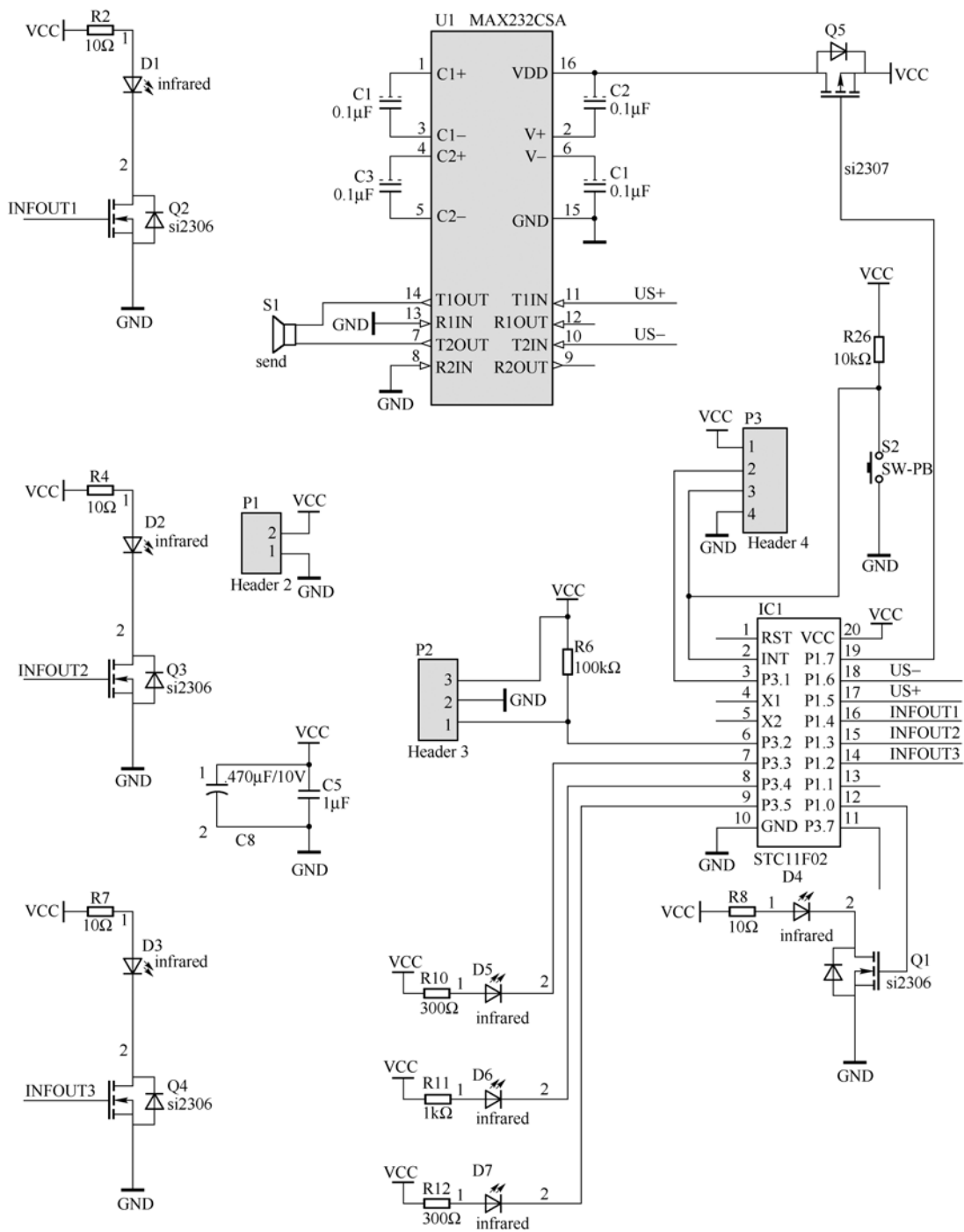


图 6-43 外部红外导航仪

6.5 软件设计

软件的开发采用 Real-View MDK ARM，开发工具为 Keil 集成环境。仿真器采用 j_link 的 USB 仿真器。STM32 的开发和以前的 ARM 7 的开发有很大的不同，芯片厂商提供了软件库，这样用户可以直接在提供的软件库中开发，带来的积极效果是方便开发，存在的问题是没有直接操作寄存器效率高。但由于 Cortex 的寄存器数量远大于 ARM 7，因此直接操作寄存器比较困难，建议还是在库的基础上开发。因为 Cortex 的速度比较快，所以整体性能还是在可接受的范围内。

6.5.1 关于 STM32 固件库

STM32 标准外设库之前的版本称为固件函数库或简称固件库，是一个固件函数包，它由程序、数据结构和宏组成，包括了微控制器所有外设的性能特征。该函数库还包括每一个外设的驱动描述和应用实例，为开发者访问底层硬件提供了一个中间 API，通过使用固件函数库，无须深入掌握底层硬件细节，开发者就可以轻松应用每一个外设。因此，使用固件函数库可以大大减少用户的程序编写时间，进而降低开发成本。每个外设驱动都由一组函数组成，这组函数覆盖了该外设的所有功能。每个器件的开发都由一个通用 API（Application Programming Interface，应用编程界面）驱动，API 对该驱动程序的结构、函数和参数名称都进行了标准化。

使用标准外设库进行开发，最大的优势在于可以使开发者不用深入了解底层硬件细节就可以灵活规范地使用每一个外设。标准外设库覆盖了从 GPIO 到定时器，再到 CAN、I²C、SPI、UART 和 ADC 等的所有标准外设。对应的 C 源代码只是用了最基本的 C 编程的知识，所有代码均经过严格测试，易于理解和使用，并且配有完整的文档，非常方便进行二次开发和应用。

6.5.2 异常信号的处理

所谓异常，指的是外界障碍物引起的、打断正常工作的事件。如果没有异常，拖地机的开发将非常简单。正是由于异常的存在，使得全覆盖清扫的目标变得复杂化。

按照前面设计的机械机构，定义拖地机的异常变量如下。

变量长度是 1 个字，从低到高的比特分别定义为：

- (0) 左边碰上；
- (1) 右边碰上；
- (2) 左边预警；
- (3) 右边预警；
- (4) 左边前方空；
- (5) 右边前方空；
- (6) 左边电动机过流；
- (7) 右边电动机过流；

- (8) 倾斜角度异常；
- (9) 行走发生顺时针侧向滑动；
- (10) 左边电动机空载；
- (11) 右边电动机空载；
- (12) 行走发生逆时针侧向滑动。

将以上异常分组，可以分为三个组：外部输入信号引起的异常，陀螺仪引起的异常，电流检测电路引起的异常。

其中，外部输入信号异常是直接和外界环境打交道的传感器引起的异常，是输入口电平变化就能检测的异常。陀螺仪异常是指由于拖地机姿态发生的变化和预期不符合引起的异常，需要使用程序进行判断才能获得，比较复杂。电流检测电器异常是指电动机是否空载或者过流，空载表示拖地机某个轮子被架空，过流表示拖地机轮子被卡死，并且前方的碰撞轻触开关并没有触发，比如用手按住拖地机上方，就会出现这个问题。

应该对拖地机的各种异常进行全面考虑，才能确保拖地机稳定工作。如果有一个没有考虑到，就可能出现在使用中出现无法正常工作情况，客户就会认为产品有问题。

1. 外部输入传感器信号

对于外部输入信号异常，下面针对硬件设计，把异常和软件连接起来。CPU 只要获得标号对应的 GPIO 的输入，就知道目前是否发生了异常、异常是什么类别，如表 6-2 所示。

表 6-2 外部输入传感器信号异常

种 类	端 口	标 号	备 注
左前碰撞	P7	XC2	检测和左前方障碍物的碰撞
右前碰撞	P4	XC1	检测和右前方障碍物的碰撞
左下空	P3	XW1	检测是否在楼梯边上
右下空	P5	XW2	检测是否在楼梯边上
左边测距报警	P3	RCV3	距离通过 CPLD 转换为数字，由 I ² C 方式读出。用来预警拖地机侧边和障碍物之间的距离，供算法使用
右边测距报警	P5	RCV4	

2. MPU 6050 陀螺仪传感器信号

MPU 6050 包含 3 轴加速度和 3 轴陀螺仪，因此能检测到很多情况。程序应该实时检测陀螺仪的数据，从而及时发现异常类别。

陀螺仪的异常种类如下。

- 轮子侧向滑动异常。在正常情况下，拖地机侧向转动时是均匀的速度，当遇到侧向有障碍物时，因为受力不平衡轮子发生侧滑，这时陀螺仪的输出立即发生角速度的变化。程序一旦判别出角速度变化，就执行一个子程序，该子程序判断是因为细微的抖动引起的变化还是真正有障碍物产生了变化，如果是障碍物引发的变化，就标记为异常。
- 由静变动的异常。原来拖地机平稳运行，突然被撞了一下，比如侧面被踢了一脚，通过对加速度阈值的设置可以知道目前是否出现这类情况。
- 水平角度的异常。比如拖地机走到一个突起的地方，机身发生侧向倾斜。可以检测

三个方向的加速度，然后计算出水平角度，就能知道是否发生了倾斜。虽然拖地机前方有两个检测掉落的传感器，但对于走上地毯这类问题则无法检测出来，而通过此方法可以顺利检测。

3. 电动机电流异常信号

电动机电流通过电阻取样由运放输出，进入 CPU 的 A/D 转换器，转换后要软件滤波，去除干扰，一旦发生过流或者空载情况，就会立刻停止拖地机的运行。对于过流，应该让拖地机后退；对于空载，说明有一个电动机输出的轮子架空，只能尝试让另外一个轮子后退，如果后退几次后异常消失，说明调整正常，否则说明无法排除问题，需要报警提示用户。

以上提到的是运行过程中的异常，还有一个是电压异常，电压异常的处理方法如下。

(1) 如果电压低于一定值但还在可以工作的范围内，则通过拖地机上的 LED 闪烁提示用户电压不足。

(2) 如果电压过低，则继续运行会损坏电池，应该立即停止。

6.5.3 电动机控制部分

按照程序要求行走的距离，根据轮子的周长计算出应当计数的脉冲数目，CPU 发出相应的 PWM 波控制电动机快慢来调节前进速度，测速传感器将电动机输出光栅上的脉冲数发送到 CPU，当 CPU 统计到规定的脉冲数量后，停止电动机运转。PWM 预置数值有快速模式、正常模式和慢速模式。

电动机可以用专门的集成电路控制正反转，但大部分消费电子产品上电动机的控制都采用分立器件，这样成本低，并且功能灵活。考虑到本拖地机的特点，不适合用普通的玩具电动机驱动芯片，因为工作电压比较高，供电电压约为 8.4V，一般的专用芯片耐压低。而用专用芯片价格偏高，并且只能按照其规定的几种方式运行。由于需要进行正反转控制，因此使用 H 桥，当采用上下桥臂上的对管同时工作的模式时，电动机控制比较重要的是死区时间。经过测试发现，mint5200 采用的就是这种模式，即上下管交替工作方式。如果不设死区时间，会瞬间直通，导致上下对管很快烧毁。这和变频调速器输出 IGBT 功率管设置死区时间是相同的道理。

只是本设计采用的并非上下双管交替工作方式，也就是正转时只导通图 6-40 中的 Q8 和 Q11，Q7 和 Q10 始终关闭；反转时导通 Q7 和 Q10，Q8 和 Q11 关闭。停止时 Q10 和 Q11 开通，Q8 和 Q7 关闭，这也具有刹车功能。因为电动机电枢电流通过 Q10 和 Q11 构成回路。经过测试，发现刹车效果非常明显。如果不刹车，只是断开电动机供电，则由于惯性，拖地机会滑行一段不希望的距离。

定义电动机控制寄存器 sr_pwmconfig 的比特的含义如下：

7	6	5	4	3	2	1	0
保留	保留	保留	右边：正反 0：正；1：反	左边：正反 0：正；1：反	运行模式设置		

最后面 3 个比特位的定义为：

bit2	bit1	bit0	模 式
0	0	0	停止
0	0	1	同时慢速
0	1	0	同时常速
0	1	1	同时快速
1	0	0	左边慢速，右边停止
1	0	1	左边中速，右边停止
1	1	0	右边慢速，左边停止
1	1	1	右边中速，左边停止

PWM1 对应 P15 的输出，速度检测对应的是 P9、CP1；PWM2 对应 P16 的输出，速度检测对应的是 P10、CP2。

电动机的正反转和刹车功能非常重要，刹车能保证停止以后不再前进，确保路线计算的精度。

这些功能在 CPLD 中用 VHDL 实现，对应的代码如下：

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity motor_mode is

port(PWM,RUN,FWD:in std_logic;
      AP,AN,BP,BN:out std_logic);
end;

architecture behav of motor_mode is
begin
  process(PWM,RUN,FWD)

    begin

    if FWD='0' and RUN='0' then
      AP<='0';
      AN<='0';
      BP<='0';
      BN<='0';
    elsif FWD='1' and RUN='0' then
      AP<='0';
      BP<='0';
      AN<='1';
      BN<='1';
    elsif FWD='1' and RUN='1' then
      AN<='0';
      BP<='0';
    end if;
  end process;
end architecture;

```

```
BN<='1';
AP<=PWM;
else
  BN<='0';
  AP<='0';
  AN<='1';
  BP<=PWM;
end if ;
end process;
end behav;
```

将以上代码创建为 quartus 中的原理图符号，并在原理图中使用它，如图 6-44 所示。分别是实体名为 motor_mode 的两个实例。为了节省 I/O 引脚，这一部分通过 SPI 方式和 CPU 连接。CPLD 电机输出引脚直接连接到 H 桥的驱动缓冲三极管。

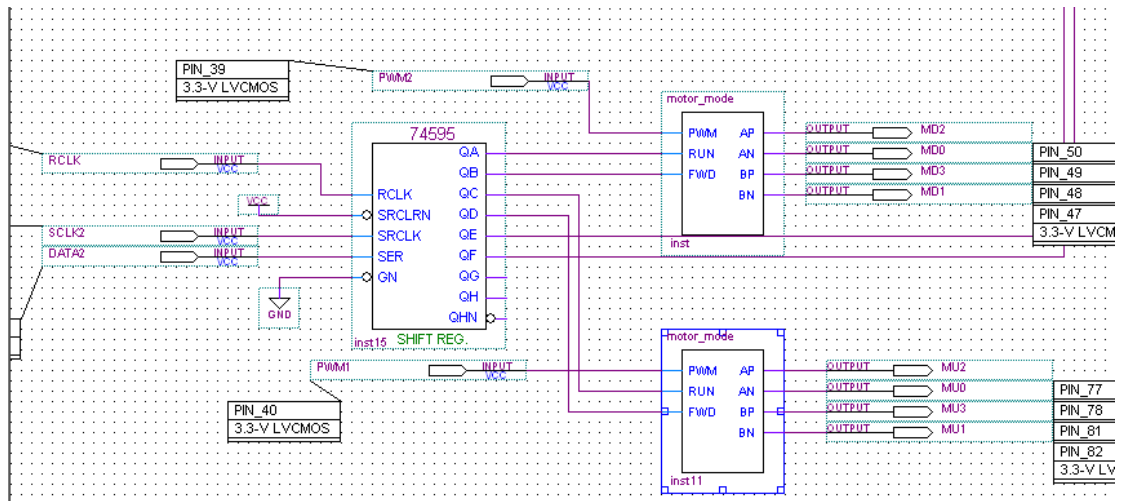


图 6-44 电动机正反转控制部分

编写软件时的控制信号和电动机工作方式关系见表 6-3。

表 6-3 控制信号和电动机工作方式关系

模 式	输 入 引 脚		输 出 引 脚			
	FWD	RUN	AP	AN	BP	BN
停止	0	0	0	0	0	0
正转	1	1	PWM	0	0	1
反转	0	1	0	1	PWM	0
刹车	1	0	0	1	0	1

6.5.4 红外和噪声波载波发生

红外采用 38kHz 载波，噪声波发射采用 40kHz 载波，用处理器直接发生载波也是可以

的，但由于处理器要处理陀螺仪实时数据，因此如果用模拟的方式产生载波会导致实时处理性能下降。所以考虑到以上因素，决定采用 CPLD 直接发生载波。这样，CPU 只要发送调制波就可以了。

下面的 VHDL 代码用来发生 38kHz 和 40kHz 的载波，这里只列出了 38kHz 的例子。首先是定义实体 S38K，然后定义端口，输入为 clk、eninfra、enultra，分别是时钟、红外允许、噪声波允许。接下来是 S38K 的结构，在 sound 的 process 敏感列表中是 clk 和 enultra，只要这两个之一发生变化，就执行 process 中的硬件逻辑。在 process 中其实是对输入时钟进行计数，达到一定的值就输出变化的电平，因此其中的数据是和时钟 clk 密切相关的，频率不同数值也不同。

```
library ieee;
use ieee.std_logic_1164.all;
entity S38K is
    port
        (   clk,eninfra,enultra    : in std_logic;
            infred38K    : out std_logic;
            s_out40k     : out std_logic
        );
end entity;
architecture rtl of S38K is
begin
    sound:    process (clk,enultra)
        variable index:integer range 0 to 300;
        variable number:integer range 0 to 30;
        begin
            if(clk'event and clk='1') then
                if(index>=201)
                    then
                        index:=0;
                        if(number<9)
                            then number:=number+1;
                        end if;
                    else index:=index+1;
                end if;
                if(enultra='1') then
                    if(index<100) then
                        if(number<9) then
                            s_out40k<='1';
                        end if;
                    else
                        s_out40k<='0';
                    end if;
                else
                    number:=0;
                end if;
            end if;
        end process;
    end architecture;
```



```

s_out40k<='0';
end if;
end if;
end process ;

```

以上代码生成一个 S38K 的原理图符号，其输出经过简单的逻辑门电路用 4 个引脚输出，分别是图 6-45 中的 PIN_97、PIN_98 和 PIN_28、PIN_29。它们工作时极性相反，经过外部驱动电路驱动噪声波发射头工作。载波的调试信号是从 enultra 输入的。

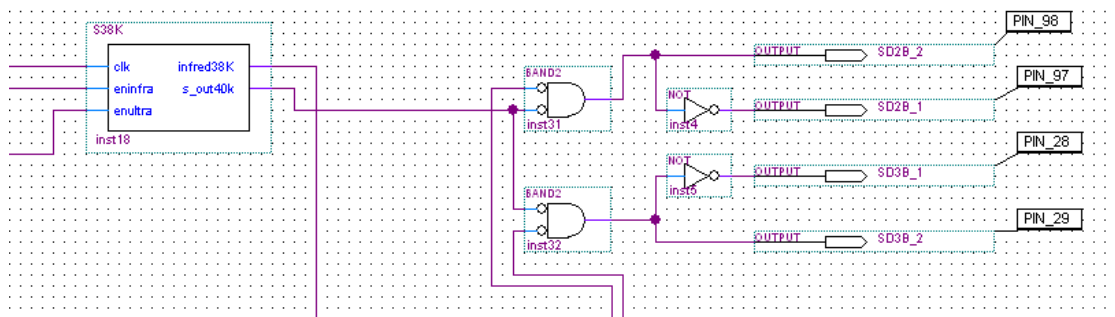


图 6-45 噪声波发射部分的 CPLD 原理图

有关红外解码器的 CPLD 实现代码及其他的 CPLD 实现代码，由于内容比较复杂，并且本章的主要目的是讲解 ARM 产品设计，包括设计思路和步骤及用到的知识，与产品本身密切相关的过于琐碎的技术实现，限于篇幅不再深入讲解。

6.5.5 PID 电动机速度控制

电动机的速度控制可采用 PID 调节器实现，PID 的原理在经典自动控制中有所介绍。考虑到累计误差的问题，一般采用增量式 PID 控制器来调节，在计算机中要对模拟方程式进行离散化操作。为了介绍 PID 电动机控制，下面先介绍过程控制的概念，然后再介绍 PID 控制。

PID 控制属于过程控制的概念。过程控制是对生产过程中的某一个或某些物理参数进行的自动控制。图 6-46 是基本的模拟 PID 控制回路。

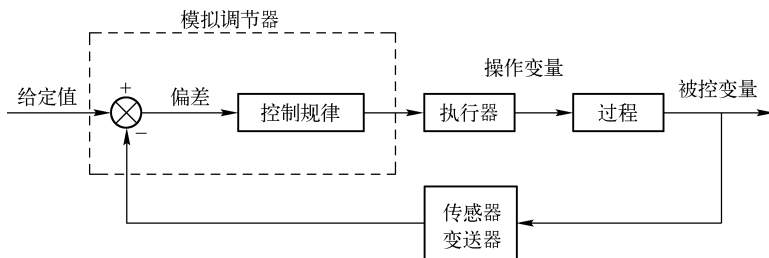


图 6-46 基本模拟 PID 控制回路

被控变量的值由传感器或变送器来检测，这个值与给定值进行比较，得到偏差，模拟

调节器根据一定的控制规律使操作变量变化,以使偏差趋近于零,其输出通过执行器作用于过程。

控制规律用对应的模拟硬件来实现,控制规律的修改需要更换模拟硬件。

目前模拟系统用得非常少,大部分采用计算机进行过程控制,图 6-47 就是用计算机代替模拟调节器进行控制的基本框图。

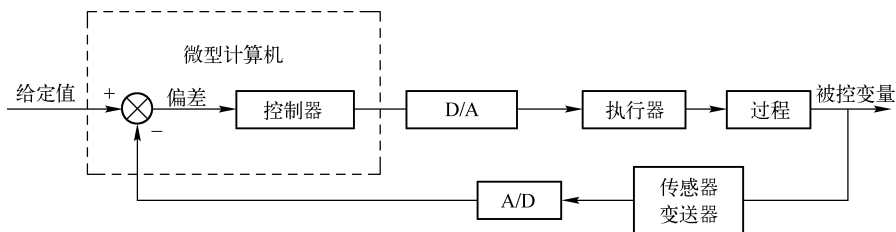


图 6-47 微机过程控制系统基本框图

借助计算机实现控制器的控制规律,是通过软件来完成的。要改变控制规律,只需改变相应的程序即可。

在介绍了过程控制系统后,下面来了解 PID 调节器的含义。模拟 PID 控制系统原理框图见图 6-48。

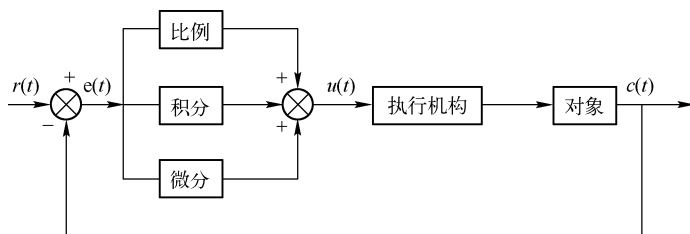


图 6-48 模拟 PID 控制系统原理框图

首先介绍模拟 PID 调节器的微分方程和传输函数。

PID 调节器是一种线性调节器,它将给定值 $r(t)$ 与实际输出值 $c(t)$ 的偏差的比例 (P)、积分 (I)、微分 (D) 通过线性组合构成控制量,对控制对象进行控制。

(1) PID 调节器的微分方程:

$$u(t) = K_p \left[e(t) + \frac{1}{T_i} \int_0^t e(t) dt + T_D \frac{de(t)}{dt} \right]$$

式中 $e(t) = r(t) - c(t)$ 。

(2) PID 调节器的传输函数:

$$D(S) = \frac{U(S)}{E(S)} = K_p \left[1 + \frac{1}{T_i S} + T_D S \right]$$

PID 调节器各校正环节的作用如下。

- 比例环节: 即时成比例地反应控制系统的偏差信号 $e(t)$, 一旦产生偏差, 调节器立即产生控制作用以减小偏差。

- 积分环节：主要用于消除静差，提高系统的无差度。积分作用的强弱取决于积分时间常数 T_I ， T_I 越大，积分作用越弱，反之则越强。
- 微分环节：能反应偏差信号的变化趋势（变化速率），并能在偏差信号的值变得太大之前，在系统中引入一个有效的早期修正信号，从而加快系统的动作速度，减小调节时间。

由于目前使用的是计算机控制 PID，因此需要修改模拟 PID 为数字 PID 控制器，也就是对模拟 PID 进行离散化，见表 6-4。

表 6-4 模拟 PID 控制规律的离散化

模拟形式	离散化形式
$e(t) = r(t) - c(t)$	$e(n) = r(n) - c(n)$
$\frac{de(t)}{dt}$	$\frac{e(n) - e(n-1)}{T}$
$\int_0^t e(t) dt$	$\sum_{i=0}^n e(i)T = T \sum_{i=0}^n e(i)$

经过推导可以得出数字 PID 控制器的差分方程为：

$$u(n) = K_p \left\{ e(n) + \frac{T}{T_I} \sum_{i=0}^n e(i) + \frac{T_D}{T} [e(n) - e(n-1)] \right\} + u_0$$

$$= u_p(n) + u_i(n) + u_d(n) + u_0$$

式中 $u_p(n) = K_p e(n)$ ，称为比例项；

$$u_i(n) = K_p \frac{T}{T_I} \sum_{i=0}^n e(i)，称为积分项；$$

$$u_d(n) = K_p \frac{T_D}{T} [e(n) - e(n-1)]，称为微分项。$$

一般而言，常用的控制方式主要有以下几种。

- P 控制： $u(n) = u_p(n) + u_0$ 。
- PI 控制： $u(n) = u_p(n) + u_i(n) + u_0$ 。
- PD 控制： $u(n) = u_p(n) + u_d(n) + u_0$ 。
- PID 控制： $u(n) = u_p(n) + u_i(n) + u_d(n) + u_0$ 。

通常有两种 PID 控制，如下所示。

(1) 位置型控制：

$$u(n) = K_p \left\{ e(n) + \frac{T}{T_I} \sum_{i=0}^n e(i) + \frac{T_D}{T} [e(n) - e(n-1)] \right\} + u_0$$

(2) 增量型控制：

$$\Delta u(n) = u(n) - u(n-1)$$

$$= K_p [e(n) - e(n-1)] + K_p \frac{T}{T_I} e(n) + K_p \frac{T_D}{T} [e(n) - 2e(n-1) + e(n-2)]$$

由于本产品是控制直流电动机的速度，也就是控制占空比，因此属于位置型控制，如图6-49所示。假如用步进电动机，则可采用增量型。

位置型的递推形式为：

$$u(n) = u(n-1) + \Delta u(n) \\ = u(n-1) + a_0 e(n) + a_1 e(n-1) + a_2 e(n-2)$$

$$\text{式中 } a_0 = K_p \left(1 + \frac{T}{T_i} + \frac{T_D}{T} \right);$$

$$a_1 = -K_p \left(1 + \frac{2T_D}{T} \right);$$

$$a_2 = -K_p \frac{T_D}{T}。$$

PID 控制需要考虑对控制量的限制，这是因为：

- (1) 控制算法总是受到一定运算字长的限制。
- (2) 执行机构的实际位置不允许超过上（或下）极限。即

$$u(n) = \begin{cases} u_{\min} & u(n) \leq u_{\min} \\ u(n) & u_{\min} < u(n) < u_{\max} \\ u_{\max} & u(n) > u_{\max} \end{cases}$$

在本产品的设计中，电动机占空比不能小于最小值，否则电动机因力矩不够而不会转动；电动机占空比也不能大于最大值，否则电动机电压过大，超过额定电压，容易损坏。

在 PID 程序设计完成后，还需要对数字 PID 控制的参数进行选择。数字 PID 参数的原则要求和整定方法如下。

(1) PID 参数原则要求：被控过程是稳定的，能迅速和准确地跟踪给定值的变化，超调量小，在不同干扰下系统输出应能保持在给定值；操作变量不宜过大，在系统与环境参数发生变化时控制应保持稳定。

显然，要同时满足上述各项要求是困难的，因此必须根据具体过程的要求，满足主要方面，并兼顾其他方面。

(2) PID 参数整定方法：

理论算法——依赖被控对象准确的数学模型（一般较难做到）。

工程整定法——不依赖被控对象准确的数学模型，直接在控制系统中进行现场整定（简单易行）。

结合本产品的特点，可以采用工程整定法。这样，只要在产品的发展样机上整定一次就可以，由于量产的产品工作环境一样，所以生产时参数不需要改动。

下面的程序片段是本拖地机一个电动机的 PID 控制子程序。

```
u16 SpeedControl()
{
    signed int Ae0=0, Be0=0; //当前偏差
```

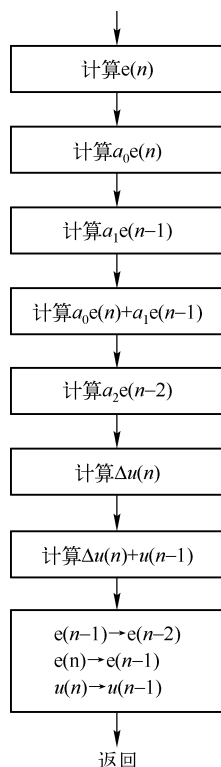


图 6-49 位置型 PID 流程

```

static signed int  Ae1=0,Be1=0;//上一次偏差
static signed int  Ae2=0,Be2=0;//上上次偏差
u16  kl;
float kc=0.5,ki=0.2,kd=0.02;
static signed int y1=0,y2=0;
{
    Ae0= (lrspeed.leftgivenspeed-CP1_Freq);
    if(abs(Ae0<1000) ) kl=1;
    else kl=1;
    y1=y1+ kc*(Ae0-Ae1)+ ki*kl*Ae0+kd*(Ae0-2*Ae1+Ae2);
    if(y1>(PWMPERIOD-20)) y1=PWMPERIOD-20;
    if(y1<PWMPERIOD/8) y1=PWMPERIOD/8;
    spd_tag1=0;
    Ae2=Ae1;
    Ae1=Ae0 ;
    TIM_OCInitStructure.TIM_Pulse = y1; //改变占空比
    TIM_OC1Init(TIM3, &TIM_OCInitStructure); //输出发生改变
}

```

6.5.6 MPU6050 陀螺仪及姿态解算

拖地机中可以用单轴陀螺仪，如 mint5200 中用的就是模拟单轴陀螺仪，但模拟陀螺仪要外接运算放大器，还需要 CPU 的 A/D 转换器进行高速处理。因为计算的是角速度，因此运算实时更新要求很高。MPU6050 是数字 3 轴陀螺仪和 3 轴加速度传感器一体的芯片，输出是 16 位的数字信号。其售价也不贵，2012 年的价格约为 14 元人民币，而 ex3500 的价格约为 8 元，加上外接的运放及多出来的焊点的贴片费用，两者价格相差不多。但是使用 MPU6050 在性能上了一个档次。

MPU6050 是美国 Invensense 的产品，MPU6000（6050）的角速度全格感测范围为 ± 250 、 ± 500 、 $\pm 1\,000$ 和 $\pm 2\,000^\circ/\text{s}$ (dps)，可准确测量快速与慢速转动。并且，用户可设置参数让加速度的测量范围达到 $\pm 2g$ 、 $\pm 4g$ 、 $\pm 8g$ 和 $\pm 16g$ 。

读取原始数据并不复杂，目前有很多基于 arduino 的程序，而在 STM32 上运行的并不多见。在 STM32 上运行需要通过 I²C 总线读取内部的数据，然后通过 CPU 定时中断程序读取，从而计算出角速度。如果计算单轴的数据，直接通过定时中断程序中的积分可以得出角度，但实验发现，如果拖地机完全水平，左右转动的计算是正确的；但如果带一定的倾斜角度转动，则实际转动的角度和计算的角度有差异，具体差异受转动速度和倾斜角度影响。原因是如果未处于水平状态，则在 x、y、z 轴中的其他两个轴具有投影分量，直接影响水平轴的计算。为了解决这个问题，需要用到旋转矩阵，通过旋转矩阵计算出欧拉角，也就是 pitch、yaw、roll。但此方法也有一定的局限性，那就是在 90° 时有参数趋向无穷大，也就是万向节锁的问题。因而改进的方法是四元素法，当然四元素法比较复杂，涉及的知识较多。在 MPU6050 内部有一个 DMP，能直接输出四元素和欧拉角。但 DPM 并不公开，需要和 Invensense 公司签约。DMP 运行前需要加载运行一个固件，也就是在初始化时加载固

件,然后 MPU6050 开始运行,并输出四元素。如果 DPM 不运行,则 MPU6050 的很多优势并不能很好得发挥。如果不用 DMP,则需要用外部处理器高速处理旋转矩阵,并在每 5ms 更新一次矩阵的计算,而且矩阵的每个元素包含了一系列不同的运算,因此对处理器资源的占用非常大。

由于单独的陀螺仪计算出来的数据也存在偏差的问题,为了使偏差最小,应该融合加速度传感器的数据,然后分别给予两者不同的权重,最终计算出经过融合的角度数据。

由陀螺仪过来的数据存在噪声,如机器运行时的细微震动、因为 PWM 波导致芯片受到的电源干扰,以及陀螺仪自身的漂移等。如果这些干扰不去除,最终计算出来的角度也不可靠,从拖地机运行的角度看就是角度调整不稳定,而且容易发生偏离。因此必须进行专门的滤波处理。可以参考卡尔曼滤波的设计,即通过卡尔曼滤波算法,将最终采集的数据收敛到和实际运行符合的曲线上。

另外,拖地机上采用陀螺仪的目的是对运行偏移角度进行校正,但是如果要求对长时间的角度偏移进行修正,则难度很大。如在 10min 内,要求角度偏移小于 2° 。陀螺仪对短时间的角度变化是能检测出来的,但对长时间的角度变化检测则很困难。因为首先陀螺仪自身的数据漂移会随着时间的积累最终反映在角度的变化上;另外,还受到 MEMS 本身最高输出精度的限制。MPU6050 具有 ± 250 、 ± 500 、 $\pm 1\,000$ 和 $\pm 2\,000^{\circ}/s$ (dps) 4 个范围,可选择设置为 $250^{\circ}/s$ 一档,这样,当 FS_SEL=0 时,在 A/D 为 16b 的输出情况下,LSB 是 $131^{\circ}/s$ 。陀螺仪还有零点漂移,而且零点漂移会一直变化。除了零点漂移外,其检测的角速度也会随着时间而漂移,并且与温度也有关系。以上这些因素都要进行深入的分析,要非常仔细地计算每个干扰因素对整体输出的影响,采取对策使输出尽可能地符合设计要求。

以下是 MPU6050 读取原始数据的程序,首先是初始化。

在初始化时将陀螺仪和加速度传感器调整为最灵敏,即 $\pm 2g$ 和 $\pm 250^{\circ}/s$,设置时钟源为 X Gyro 参考输入,这比内部振荡器的性能稍微好一些。

```
void MPU6050_Initialize()
{
    MPU6050_SetClockSource(MPU6050_CLOCK_PLL_ZGYRO);
    MPU6050_SetFullScaleGyroRange(MPU6050_GYRO_FS_250);
    MPU6050_SetFullScaleAccelRange(MPU6050_ACCEL_FS_2);
    MPU6050_SetSleepModeStatus(DISABLE);
}
```

接下来测试 MPU6050 是否连接正常,主要检测焊接贴片是否有问题。

```
bool MPU6050_TestConnection()
{
    if(MPU6050_GetDeviceID() == 0x34) //0b110100; 8-bit representation in hex = 0x34
        return TRUE;
    else
        return FALSE;
}
```

然后在定时中断程序内读取原始输出数据。下列程序读出三个方向的加速度和三个方向的角速度数据：

```
void MPU6050_GetRawAccelGyro(s16* AccelGyro)
{
    int i;
    u8 tmpBuffer[14];
    MPU6050_I2C_BufferRead(MPU6050_DEFAULT_ADDRESS,tmpBuffer,
MPU6050_RA_ACCEL_XOUT_H, 14);
    /* Get acceleration */
    for( i=0; i<3; i++)
        AccelGyro[i]=((s16)((u16)tmpBuffer[2*i] << 8) + tmpBuffer[2*i+1]);
    /* Get Angular rate */
    for( i=4; i<7; i++)
        AccelGyro[i-1]=((s16)((u16)tmpBuffer[2*i] << 8) + tmpBuffer[2*i+1]);
}
```

一种简单的方法是在数据读出以后，经过滤波，在中断程序内通过积分计算获得角度偏移。如果是单轴陀螺仪，这样的计算在短时间内不会影响性能，但是如果要求长时间保持角度的稳定，就不容易实现了。因为拖地机运行时，特别是在转弯的时候，前部的机身角度有起伏，这是一个三维的姿态变化，需要用到前面提到的矩阵运算，所以要做更加复杂的计算。MPU6050 内部有一个 DMP，其实就是一个处理器，对原始数据进行处理。如果用了 DMP，直接输出解算好的数据就可以，外部 CPU 的负担很轻，但假如不用内部的 DMP，则还需要自己解算，对软件和硬件都是不小的负担。

陀螺仪调试正常后，实际测试结果表明，在 10min 内能让方向维持在可接受的范围，超过就不理想了。因此，拖地机还需配合外部红外线定时修正角度，具体方法见本章前面的导航部分。

直线行走的陀螺仪的角度修正在主控程序中进行，当检测到角度发生偏转后，调整左右电动机的速度，使角度恢复到偏移前的状态。

侧向滑动的角度检测也通过陀螺仪数据进行，知道了侧向滑动就能记录目前的路线，分析障碍物的情况，决定下一步的动作。

6.5.7 有关清洁覆盖算法分析

这里主要介绍简单的 Z 字形行走，如图 6-50 所示。直线行走用到了前面的角度修正，水平方向拖地机 T 按照 x 轴正方向移动。具体工作方式如下。

T 首先处于停车状态，启动以后，执行正向行驶命令 FWD，沿着 y 轴正方向前进，遇到墙壁 W1 停止，然后后退一定距离，左边单个轮子右转 180°，沿着 y 轴反方向前进。遇到 W2 停止，后退一定距离，右边单个轮子左转 180°，沿着 y 轴正方向行驶。

假设正在清扫右边区域，在向 y 轴正方向行驶时，右侧的传感器不断地检测和右边的距离，到达 W1 时，先停车，接着后退一定距离。如果距离小于 25cm，记录目前的 y 数据，如果距离小于 5cm，标记目前为中心旋转方式，否则右转 180°。如果没有记录过近的 y 数

据,则正常运行。如果记录的 y 数据一直保持为 5cm 近距离,可以认为边上是墙。这时执行中心旋转方式,左旋 45° ,左侧碰撞传感器遇到障碍,则方向回正,开始沿着墙壁直行,到达 W2 后,认为右侧清扫结束。然后沿着 W2 回到出发点,开始打扫左侧。

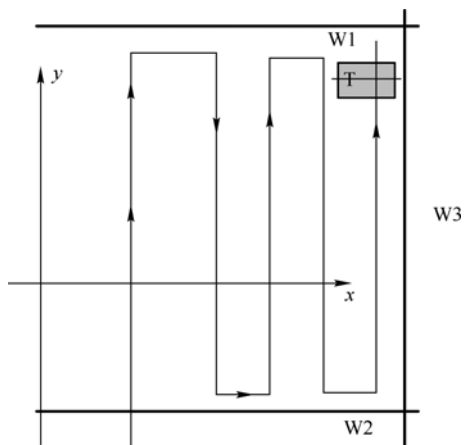


图 6-50 Z 字形行走

以上针对的是墙壁规则的情况,对应墙壁不规则的情况,如图 6-51 所示。

到达 W1 时先停车,接着后退一定距离。如果距离小于 25cm,记录目前的 y 数据,如果距离小于 5cm,标记目前为中心旋转方式,否则右转 180° 。如果没有记录过近 5cm 的 y 数据,则正常运行。否则分析 y 数据,若 y 数据并非一直相等,说明有类似图 6-51 所示的墙壁或者障碍物。 y 数据在读入的同时, x 距离数据也在读入。当 $(y_1 - y_2)$ 的差大于 T 自身的宽度时,说明 T 可以进去,因此 T 首先走到 (x_1, y_1) 处。分析深度,如果 x_1 和 x_2 的数据相差很大,如 10cm,说明有两个不同的物体;如果 x_1 和 x_2 相等,说明中间有凹进去的地方,此时可根据情况,运行垂直或者水平清扫方式。

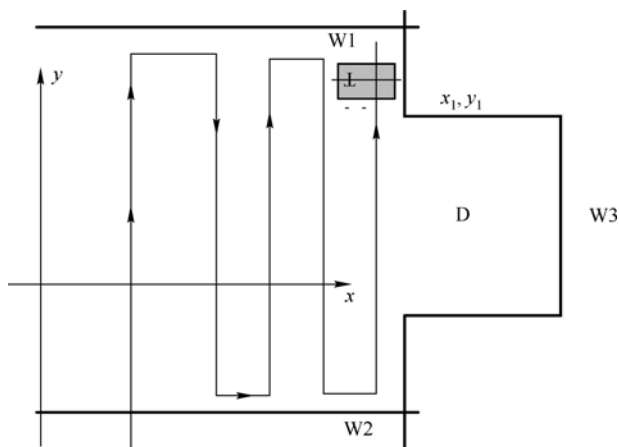


图 6-51 墙壁不规则的情况

垂直方式的要求为深度 $> 50\text{cm}$, $\Delta y > 1\text{m}$; 否则是水平模式。

当区域 D 打扫完后，回到 (x_2, y_2) 的地方，继续向 W2 走，直到结束。

对于房间内放了少量简单形状的家具的情况，分析如下。

(1) 方形家具存在的情况，如图 6-52 所示。

如果 W3 和黑色区域之间的间隙距离小于拖地机 T 本身的大小，这时 T 首先单轮右转，右侧传感器发生碰撞，说明右侧距离很小，T 恢复原状，开始沿中心右转，然后左转 45° ，前进，碰到 W3，沿中心右转 45° ，执行边沿墙壁探测边清扫的程序。

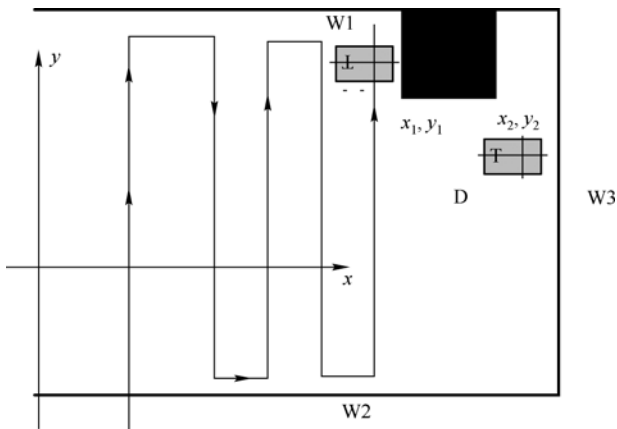


图 6-52 方形家具存在的情况

(2) 圆形障碍物的情况，如图 6-53 所示。

第一次左侧碰到圆柱，执行圆柱上方区域的迂回运行，然后，前方传感器右侧碰到，左侧未碰到，说明圆柱结束；也可能是左侧、右侧恰好都未碰到，同样说明圆柱结束。接着，T 沿着圆柱的下半部分移动到第一次遇到圆柱的位置，此时相当于圆柱的周边清扫完成。然后开始圆柱的下半部分边沿到 W2 之间的迂回操作。

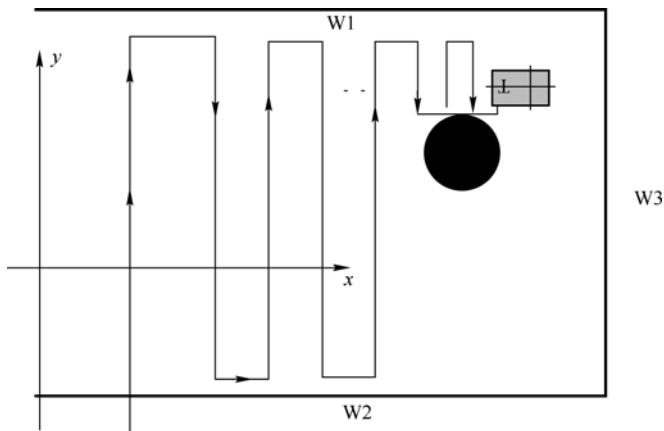


图 6-53 圆形障碍物的情况

(3) 遇到两个连续的障碍物的情况，如图 6-54 所示。

当遇到 B1 左边的凸出处时，T 的左边碰撞传感器检测到，如果是凹形的，也可能是右

边的碰撞传感器检测到, 如果是平面的, 则左右两个传感器同时检测到。记录下目前 $B1$ 的 (x_1, y_1) 。然后继续在 $B1$ 和 $W1$ 之间迂回, 当走到 $B1$ 右侧边界时, T 的右边碰撞传感器检测到, 左边碰撞传感器没有检测到, 这时记下 (x_2, y_2) 。也可能是 T 恰好在 $B1$ 最右边的边缘处。这时 T 没有碰到 $B1$ 的最右边, T 将前进到 $W2$, 然后返回 $B1$ 的 y_1 水平线处, 执行 $B1$ 下方围绕算法, 直到走到 $B1$ 的 (x_1, y_1) 处。

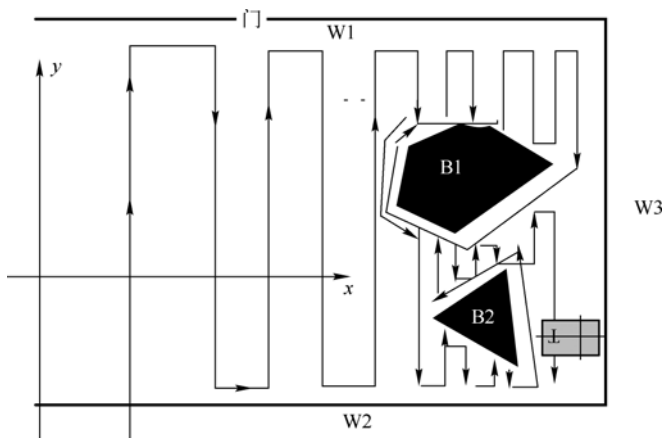


图 6-54 遇到两个连续障碍物情况

(4) 拖地机的路线表示方法, 如图 6-55 所示。

拖地机开始的方向为 x 方向, 无论怎样初始安放, x 方向都是机器头指向的方向。假设 (x, y) 的第一象限是 $10\text{m} \times 10\text{m}$, 在一个房间内, 这个工作要重复 4 次。第一象限清扫结束, 坐标旋转 90° , 开始另外 $1/4$ 面积的打扫, 直到 4 个部分全部结束。

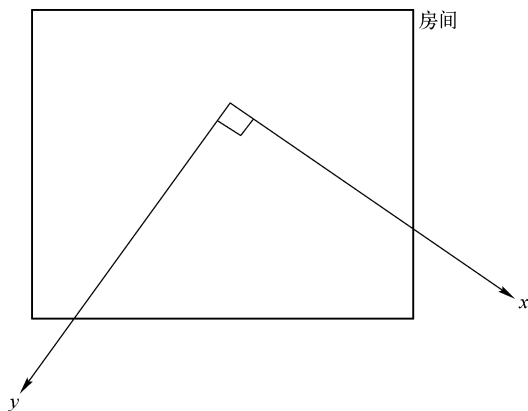


图 6-55 路线的 4 象限表示法

对在某象限内的清扫，首先要定义单位打扫面积为扫地机自身的面积，假设为 $25\text{cm} \times 25\text{cm}$ 。设计如上面介绍的绝对坐标，将整个 $10\text{m} \times 10\text{m}$ 的面积细分为一个个的小块，每个小块对应 1b （比特）。

1 字节对应 2m, 10m 对应 5 字节。

路线在内存中的表示如图 6-56 所示。

设机器自身的面积为 $25\text{cm} \times 25\text{cm}$ ， 1m^2 包含 16 个这样的面积。用 1b 表示对应面积， 1m^2 用 16b，2 字节表示。则 100m^2 为 200 字节。清扫可以在内存中表示为如图 6-56 所示，这样就不需要对坐标进行变换了，减少了运算。

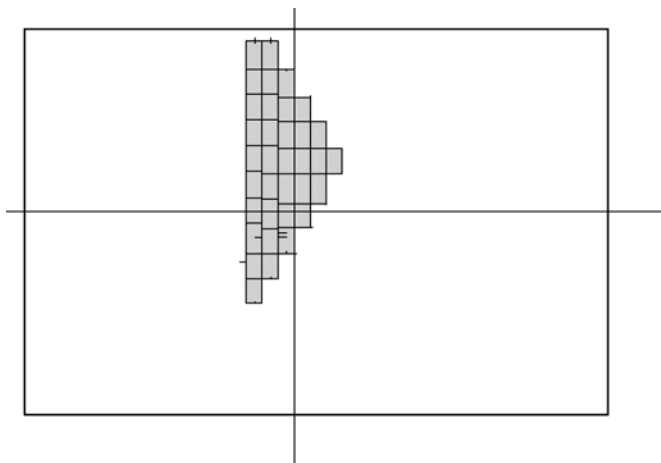


图 6-56 路线在内存中的表示

图 6-56 中栅格内存包括 4 部分，每一部分都表示为 $10\text{m} \times 10\text{m}$ ，因此一共是 800 字节。

每个格子用 1b 表示。除了这张表以外，还有一张表，和这个表完全对应，里面的比特表示这张表对应的比特是允许清扫的区域还是有物体的区域。因此 $10\text{m} \times 10\text{m}$ 总共两张表，1 600 字节。在打扫结束时，机器通过搜索这张表，沿着房间的墙壁绕一周。

6.5.8 规则动作库

拖地机的运行按照一定的规则进行。在 Z 字形运行路线中，需要在遇到障碍物后进行分析，执行相应的动作，如图 6-57 所示。也许有人要问：为什么遇到前面的障碍物要后退？因为拖地机是方的，不后退直接转弯容易卡死，这和普通的圆形扫地机不同。表 6-5 是对应的规则动作库。

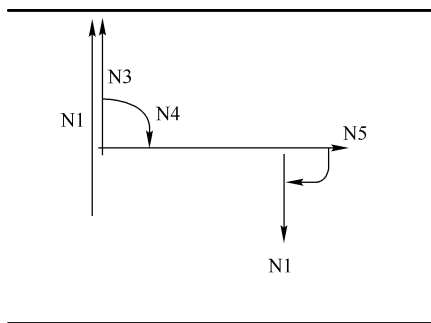


图 6-57 Z 字形避障的分析

表 6-5 规则动作库

电动机过流	左电动机过流	右前落空	左前落空	右电动机空载	左电动机空载	右碰撞	左碰撞	屏蔽码	原因，导致的下一步动作，以及下一步动作的持续时间					
									动作	原因	方向	下一步动作	持续时间	标记
7	6	5	4	3	2	1	0	0C	N1	架空		全局停止		N0
×	×	×	×	0	1	×	×							
×	×	×	×	1	0	×	×							
×	×	×	×	1	1	×	×							
0	0	0	0	0	0	0	0	FF	N1	正常	x	前进		N1
0	1	0	1	x	x	0	1	F3	N1	过流落空碰撞	x	刹车停止1s，后退10cm	1S	N3
1	0	1	0	x	x	1	0							
1	1	1	1	x	x	1	1							
0	0	0	0	0	0	0	0	FF	N3	执行完毕	F	中心右转		N4
										B	中心左转			
0	0	0	0	0	0	0	0	FF	N4	执行完毕	x	前进10cm		N5
0 1 1	1 0 1	0 1 1	1 0 1	x	x	0 1 1	1 0 1	F3	N4 Pause	前落空或者遇到碰撞，阻力	x	停止，后退5cm		N5
									N5	前进时调整电动机速度差，沿墙走	F	中心右转		N1
											B	中心左转		
									N6		x	转45°，前进10cm		N7
0	0	0	0	0	0	0	0	FF	N7	执行完毕	F	中心右转		N1
											B	中心左转		

以上动作变换采用状态机的方式编程。除了上述动作外, 还需要考虑障碍物类型的检测、沿着家具边清洁、沿着墙壁清洁, 这些都要根据陀螺仪的数据进行。由于具体程序设计过于琐碎复杂, 细节工作量非常大, 这里不再详细介绍。

6.6 拖地机产品样机

在拖地机产品的软件和硬件设计完成后, 还需要反复进行调试, 以确保产品的性能可靠稳定。这部分的工作量相当大。如果顺利可能不需要花太多时间, 如果不顺利, 这个开发时间就很长, 投资将很大, 产品开发的风险也在于此。产品开发完毕, 需要进入量产前的小规模试生产, 在没有问题的情况下进入中规模生产, 样机如图 6-58 所示。小批量产品的生产分为几个部分, 首先是塑料外壳的制造, 也就是将模具提交给注塑厂加工; 然后是贴片厂

代加工电路板，等加工完毕后再拿回自己公司组装和测试。

当然，通常从样机到真正的产品中间还要做不少修改，而且第一批出去的产品非常可能有质量问题。这些都是客观存在的事实，只能通过逐步的完善加以解决。要想确保新产品第一次生产没有任何问题，可能性不大，除非有足够的资金支撑，允许做长时间的、大量的测试。因此要有足够的心理准备，以应对因为前期质量问题而引起的商业销售问题。



图 6-58 拖地机的样机

6.7 拖地机专利撰写举例

产品在设计时需要考虑知识产权的保护，特别是消费电子产品，如果销量好，则一定有人仿制。为了保护开发成果，一方面要对硬件和软件进行加密，另一方面要申请专利保护。

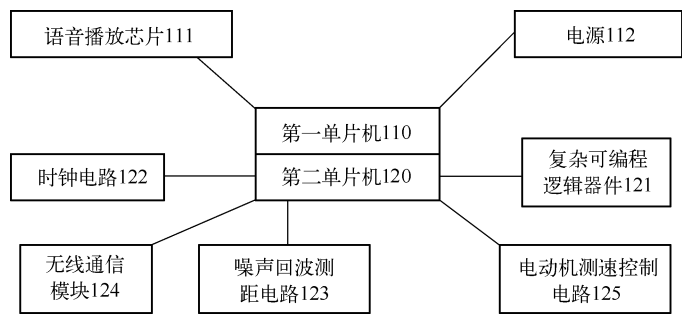
以下是本拖地机的一个实用新型专利的例子。本专利已经授权，专利号是 ZL201220480135.9,可在国家知识产权网上查询到。目前我国的实用新型专利是登记制，不进行实质审查，相对比较容易获得通过，一般一年内能获得授权。而发明专利需要进行实质审查，难度相对比较大。如果申请后的实质审查有问题，专利局将发审查意见通知书，应根据审查员发出的审查意见，提出相应的答复意见。这中间可能有多个来回，直到出现以下结果：申请获得通过，或者被驳回。

说明书摘要

本实用新型公开了一种具有导航功能的拖地机器人控制主机，它包括控制主机和导航仪。所述导航仪通过发送 3 束不同编码的红外线，主机接收红外线并分析来确定目前在房间的位置；所述控制主机的核心包括用于管理电源的第一单片机和用于完成拖地机功能控制的第二单片机，所述第二单片机通过端口连接有复杂可编程逻辑器件、时钟电路、噪声回波测距电路、无线通信模块、电动机测速控制电路。本实用新型的有益效果在于：通过导航仪导航的拖地机主机能用一种规划路径的方法完全覆盖室内的拖地面积，杜绝重复拖地的情况，使有限的内置电池电量获得最大程度的利用，大大扩展了清扫面积。从而使拖地机能快速完

成室内的清洁工作。

摘要附图



权利要求书

1. 一种具有导航功能的拖地机器人控制主机，其特征在于，它包括控制主机和导航仪，所述导航仪通过无线通信装置实现和控制主机的双向通信。
- 所述控制主机的核心包括用于管理电源的第一单片机和用于完成拖地机功能控制的第二单片机。所述第二单片机通过端口连接有：
- 复杂可编程逻辑器件，用于实现控制主机的外设扩展；
- 时钟电路，用于实现控制主机的内部自动计时；
- 噪声回波测距电路，用于实时检测拖地机器人和周围环境的距离信息，并将其反馈给单片机；
- 无线通信模块，用于实现第二单片机和导航仪的无线通信；
- 电动机测速控制电路，用于实时检测拖地机器人驱动电动机的转速信息，并将其反馈给单片机。
2. 根据权利要求 1 所述的一种具有导航功能的拖地机器人控制主机，其特征在于，所述无线通信模块包括无线通信集成电路和红外通信模块。
3. 根据权利要求 1 所述的一种具有导航功能的拖地机器人控制主机，其特征在于，所述控制主机连接有外设的控制面板和液晶显示器，所述液晶显示器通过 SPI 接口连接控制主机。
4. 根据权利要求 3 所述的一种具有导航功能的拖地机器人控制主机，其特征在于，所述控制面板设有电源按钮、干拖按钮和湿拖按钮。
5. 根据权利要求 1 所述的一种具有导航功能的拖地机器人控制主机，其特征在于，所述控制主机还设有与第二单片机连接的语音播放芯片。
6. 根据权利要求 1 所述的一种具有导航功能的拖地机器人控制主机，其特征在于，所述电动机测速控制电路包括红外反射型传感器，红外反射型传感器安装于电动机转子上的测速盘处，用于检测电动机的转速信息。红外通信模块与第二单片机连接，用于接收导航编码

信息并将其传输给第二单片机。

7. 根据权利要求 1 所述的一种具有导航功能的拖地机器人控制主机, 其特征在于, 所述拖地机器人的电源采用大容量锂电池, 第一单片机通过充电控制电路连接锂电池。

说明书

一种具有导航功能的拖地机器人控制主机

技术领域

本实用新型涉及机器人控制领域, 具体地说, 特别涉及一种具有导航功能的拖地机器人控制主机。

背景技术

拖地机器人能替代人工劳动, 全自动全覆盖清扫房间。拖地机器人是依靠内部的计算机控制系统, 利用各种传感器感知外界的环境, 绕过障碍物, 并对行走的路线进行合理规划, 在减少重复清扫的前提下实现室内清扫面积的全覆盖。目前大部分拖地机器人都是采用随机路线清扫, 重复清扫的区域很大, 浪费了宝贵的电池电力, 从而使续航里程大大缩短。

实用新型内容

本实用新型的目的在于提供一种具有导航功能的拖地机器人控制主机, 通过采用控制主机+导航仪架构的方案, 控制主机完成拖地机器人的基本输入/输出操作, 以及清扫工作的基本面积单元, 导航仪则将房间路径分解为多个基本面积单元并将其通过无线方式下载到控制主机, 使拖地机完成对多个房间的清扫, 并避免重复清扫。

本实用新型克服了传统技术中的不足, 从而实现本实用新型的目的。

本实用新型所解决的技术问题可以采用以下技术方案来实现:

一种具有导航功能的拖地机器人控制主机, 它包括控制主机和导航仪, 所述导航仪通过无线通信装置实现与控制主机的双向通信;

所述控制主机的核心包括用于管理电源的第一单片机和用于完成拖地机功能控制的第二单片机。所述第二单片机通过端口连接有:

复杂可编程逻辑器件, 用于实现控制主机的外设扩展;

时钟电路, 用于实现控制主机的内部自动计时;

噪声回波测距电路, 用于实时检测拖地机器人和周围环境的距离信息, 并将其反馈给单片机;

无线通信模块, 用于实现第二单片机和导航仪的无线通信;

电动机测速控制电路, 用于实时检测拖地机器人驱动电动机的转速信息, 并将其反馈给单片机。

在本实用新型的一个实施例中, 所述无线通信模块包括无线通信集成电路和红外通信模块。

在本实用新型的一个实施例中, 所述控制主机连接有外设的控制面板和液晶显示器,

所述液晶显示器通过 SPI 接口连接控制主机。

进一步,所述控制面板设有电源按钮、干拖按钮和湿拖按钮。

在本实用新型的一个实施例中,所述控制主机还设有与第二单片机连接的语音播放芯片。

在本实用新型的一个实施例中,所述电动机测速控制电路包括红外反射型传感器。红外反射型传感器安装于电动机的转子处,用于检测电动机的转速信息;红外通信模块与第二单片机连接,用于接收导航编码信息并将其传输给第二单片机。

在本实用新型的一个实施例中,所述拖地机器人的电源采用大容量锂电池,第一单片机通过充电控制电路连接锂电池。

本实用新型的有益效果在于:通过导航仪导航的拖地机主机能用一种规划路径的方法完全覆盖室内的拖地面积,杜绝重复拖地的情况,使有限的内置电池电量获得最大程度的利用,大大扩展了清扫面积。从而使拖地机能快速完成室内的清洁工作。

附图说明

图 1 为本实用新型所述的拖地机器人的结构框图。

图 2 为本实用新型所述的控制主机的结构框图。

具体实施方式

为使本实用新型实现的技术手段、创作特征、达成目的与功效易于理解,下面结合具体实施方式,进一步阐述本实用新型。

如图 1 所示,本实用新型所述的一种具有导航功能的拖地机器人控制主机,它包括控制主机 100 和导航仪 200。所述导航仪通过无线通信装置实现和控制主机的双向通信,控制主机 100 完成拖地机器人的基本输入/输出操作,以及清扫工作的基本面积单元,导航仪 200 则将房间路径分解为多个基本面积单元并将其通过无线方式下载到控制主机,使拖地机完成对多个房间的清扫,并避免重复清扫。

如图 2 所示,所述控制主机的核心包括第一单片机 110 和第二单片机 120。其中,第一单片机用于管理电源,第二单片机用于完成拖地机功能控制。所述第二单片机通过端口连接有:

复杂可编程逻辑器件 121,用于实现控制主机的外设扩展;

时钟电路 122,用于实现控制主机的内部自动计时;

噪声回波测距电路 123,用于实时检测拖地机器人和周围环境的距离信息,并将其反馈给单片机;

无线通信模块 124,用于实现第二单片机和导航仪的无线通信;

电动机测速控制电路 125,用于实时检测拖地机器人驱动电动机的转速信息,并将其反馈给单片机。

在本实用新型的一个实施例中,所述无线通信模块包括无线通信集成电路和红外通信模块。

在本实用新型的一个实施例中,所述控制主机连接有外设的控制面板和液晶显示器,

所述液晶显示器通过 SPI 接口连接控制主机。

进一步，所述控制面板设有电源按钮、干拖按钮和湿拖按钮。

在本实用新型的一个实施例中，所述控制主机还设有与第二单片机连接的语音播放芯片 111。

尤其需要指出的是，所述电动机测速控制电路包括红外反射型传感器和红外通信模块，红外反射型传感器安装于电动机转子上的测速盘处，用于检测电动机的转速信息；红外通信模块与第二单片机连接，用于接收导航编码信息并将其传输给第二单片机。

在本实用新型的一个实施例中，所述拖地机器人的电源 112 采用大容量锂电池，第一单片机通过充电控制电路连接锂电池。

本实用新型通过导航仪导航的拖地机主机能用一种规划路径的方法完全覆盖室内的拖地面积，杜绝重复拖地的情况，使有限的内置电池电量获得最大程度的利用，大大扩展了清扫面积。从而使拖地机能快速完成室内的清洁工作。

以上显示和描述了本实用新型的基本原理与主要特征及本实用新型的优点。本行业的技术人员应该了解，本实用新型不受上述实施例的限制，上述实施例和说明书中描述的只是说明本实用新型的原理，在不脱离本实用新型精神和范围的前提下，本实用新型还会有各种变化和改进，这些变化和改进都落入要求保护的本实用新型范围内。本实用新型要求保护范围由所附的权利要求书及其等效物界定。

说明书附图

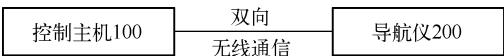


图 1

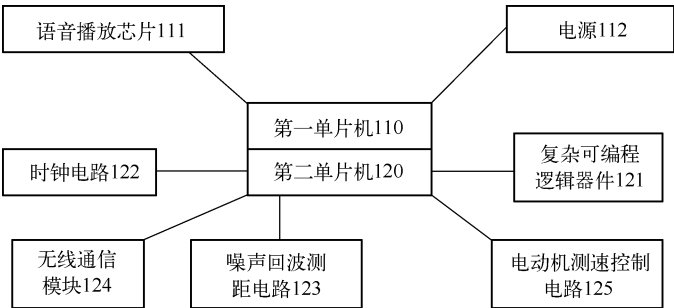


图 2

第 7 章

基于 OS 层面 ARM 必备知识—— 嵌入式 Linux 文件系统

本章首先介绍 Linux 内核对文件系统的支持，然后介绍虚拟文件系统（VFS）和其他常见的嵌入式文件系统，最后介绍 Flash 文件系统和根文件系统。这些都是移植嵌入式 linux 必备的知识。

7.1 Linux 文件系统简介

Linux 的重要特征之一就是支持多种文件系统, 这样使得它更加灵活并且可以和许多其他操作系统共存。Linux 支持 15 种文件系统: ext、ext2、xia、minix、umsdos、msdos、vfat、proc、smb、ncp、iso9660、sysv、hpfs、affs 及 ufs。毫无疑问, 今后支持的文件系统类型还将增加。

Linux 并不使用设备标志符(如设备号或驱动器名称)来访问独立文件系统, 而是通过一个将整个文件系统表示成单一实体的层次树结构来访问它。Linux 每安装(mount)一个文件系统时都会将其加入到文件系统层次树中。无论文件系统属于什么类型, 都被连接到一个目录上并且此文件系统上的文件将取代该目录中已存在的文件。这个目录称为安装点或者安装目录。当卸载此文件系统时这个安装目录中原有的文件将再次出现。

当磁盘初始化时(使用 fdisk), 磁盘中将添加一个描述物理磁盘逻辑构成的分区结构。每个分区可以拥有一个独立的文件系统, 如 ext2。文件系统将文件组织成包含目录。软连接等存在于物理块设备中的逻辑层次结构。包含文件系统的设备叫块设备。Linux 文件系统认为这些块设备是简单的线性块集合, 它并不关心或理解底层的物理磁盘结构。这个工作由块设备驱动来完成, 由它将对某个特定块的请求映射到正确的设备上去; 此块所在硬盘的对应磁道、扇区及柱面数都被保存起来。无论哪个设备持有这个块, 文件系统都必须使用相同的方式来寻找并操纵此块。Linux 文件系统不管(至少对系统用户来说)系统中有哪些不同的控制器控制着哪些不同的物理介质且这些物理介质上有几个不同的文件系统。文件系统甚至还可以不在本地系统而在通过网络连接的远程硬盘上。

文件系统上的文件是数据的集合。文件系统不仅包含文件中的数据, 而且还包含文件系统的结构, 所有 Linux 用户和程序看到的文件、目录、软连接及文件保护信息等都存储在其中。此外, 文件系统中必须包含安全信息以便保持操作系统的基本完整性。没有人愿意使用一个动不动就丢失数据和文件的操作系统。

Linux 最早的文件系统是 minix, 它受限很大且性能低下。其文件名最长不能超过 14 个字符(虽然比 8.3 文件名要好)且最大文件大小为 64MB。64MB 看上去很大, 但实际上一个中等的数据库都会超过这个大小。第一个专门为 Linux 设计的文件系统称为扩展文件系统(Extended File System)或 ext。它出现于 1992 年 4 月, 虽然能够解决一些问题但性能依旧不好。1993 年扩展文件系统第 2 版或 ext2 被设计出来并添加到 Linux 中。

将 EXT 文件系统加入 Linux 产生了重大影响。每个实际文件系统都从操作系统和系统服务中分离出来, 它们之间通过一个接口层——虚拟文件系统或 VFS 来通信。

VFS 使得 Linux 可以支持多个不同的文件系统, 每个文件系统表示一个 VFS 的通用接口。由于软件将 Linux 文件系统的所有细节进行了转换, 所以 Linux 核心的其他部分及系统中运行的程序将看到统一的文件系统。Linux 的虚拟文件系统允许用户能同时透明地安装许多不同的文件系统。

虚拟文件系统的设计目标是为 Linux 用户提供快速且高效的文件访问服务。同时它必须保证文件及其数据的正确性。这两个目标相互间可能存在冲突。当安装一个文件系统并使用

时, Linux VFS 为其缓存相关信息。此缓存中的数据在创建、写入和删除文件与目录时如果被修改, 则必须谨慎地更新文件系统对应的内容。如果能够在运行核心内看到文件系统的数据结构, 那么就可以看到那些正被文件系统读写的数据块。描述文件与目录的数据结构被不断地创建与删除, 而设备驱动将不停地读取与写入数据。这些缓存中最重要的是 Buffer Cache, 它被集成到独立文件系统访问底层块设备的例程中。当进行块存取时数据块首先被放入 Buffer Cache 里并根据其状态保存在各个队列中。此 Buffer Cache 不仅缓存数据, 而且帮助管理块设备驱动中的异步接口。

7.1.1 ext2 和 INODE

第二代扩展文件系统由 Rey Card 设计, 其目标是为 Linux 提供一个强大的可扩展文件系统。它同时也是 Linux 中设计最成功的文件系统。ext2 文件系统的物理分布见图 7-1。

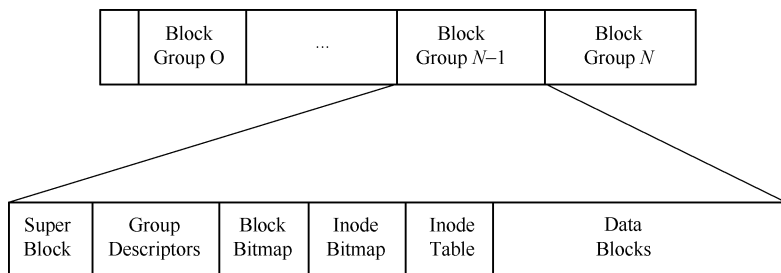


图 7-1 ext2 文件系统的物理分布

同很多文件系统一样, ext2 建立在数据被保存在数据块中的文件内这个前提下。这些数据块长度相等且这个长度可以变化, 某个 ext2 文件系统的块大小在创建(使用 mke2fs)时设置。每个文件的大小和刚好大于它的块大小的整数倍相等。如果块大小为 1 024 字节, 则一个 1 025 字节长的文件将占据两个 1 024 字节大小的块。这样不得不浪费差不多一倍的空间。我们通常需要在 CPU 的内存利用率和磁盘空间使用上进行折中。而大多数操作系统, 包括 Linux 在内, 为了减小 CPU 的工作负载而被迫选择相对较低的磁盘空间利用率。并不是文件中的每个块都包含数据, 其中有些块用来包含描述此文件系统结构的信息。ext2 通过一个 inode 结构来描述文件系统中的文件并确定此文件系统的拓扑结构。inode 结构描述文件中的数据占据哪个块和文件的存取权限、文件修改时间及文件类型。ext2 文件系统中的每个文件用一个 inode 来表示且每个 inode 有唯一的编号。文件系统中所有的 inode 都被保存在 inode 表中。ext2 目录只是一个包含指向其目录入口指针的特殊文件(也用 inode 表示)。

ext2 inode 见图 7-2。

在 ext2 文件系统中 inode 是基本块; 文件系统中的每个文件与目录由唯一的 inode 来描述。每个数据块组的 ext2 inode 被保存在 inode 表中, 同时还有一个位图被系统用来跟踪已分配和未分配的 inode。图 7-2 给出了 ext2 inode 的格式, 它包含以下几个域。

1) Mode

它包含两类信息: inode 描述的内容和用户使用权限。ext2 中的 inode 可以表示一个文件、目录、符号连接、块设备、字符设备或 FIFO。

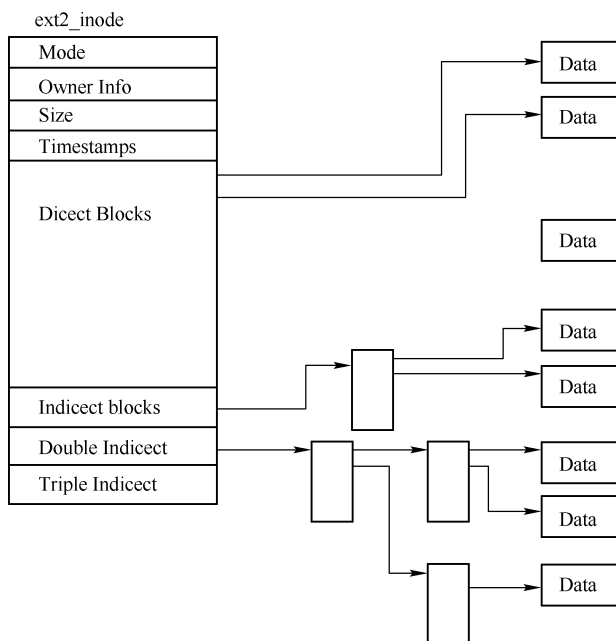


图 7-2 ext2 inode

2) Owner Info

它表示此文件或目录所有者的用户和组标志符。文件系统根据它可以进行正确的存取。

3) Size

Size 表示以字节计算的文件尺寸。

4) Timestamps

它表示 inode 创建及最后一次被修改的时间。

5) Direct Blocks

它表示指向此 inode 描述的包含数据的块指针。前 12 个指针指向包含由 inode 描述的物理块，最后 3 个指针包含多级间接指针。例如，两级间接指针指向一个块指针，而这些指针又指向一些数据块。这意味着访问文件尺寸小于或等于 12 个数据块的文件将比访问大文件快得多。

ext2 inode 还可以描述特殊设备文件。虽然它们不是真正的文件，但可以通过它们访问设备。所有那些位于/dev 中的设备文件均可用来存取 Linux 设备。例如，mount 程序可将设备文件作为参数。

7.1.2 虚拟文件系统（VFS）

图 7-3 给出了 Linux 核心中虚拟文件系统和实际文件系统间的关系。此虚拟文件系统必须能够管理在任何时刻 mount 到系统的不同文件系统。它通过维护一个描述整个虚拟文件系统和实际已安装文件系统的结构来完成这个工作。

容易让人混淆的是 VFS 使用了与 ext2 文件系统类似的方式——超块和 inode 来描述文件系统。像 ext2 inode 一样，VFS inode 描述系统中的文件和目录及 VFS 中的内容和拓扑结

构。从现在开始，我们将用 VFS inode 和 VFS 超块来把它们与 ext2 inode 和超块进行区分。

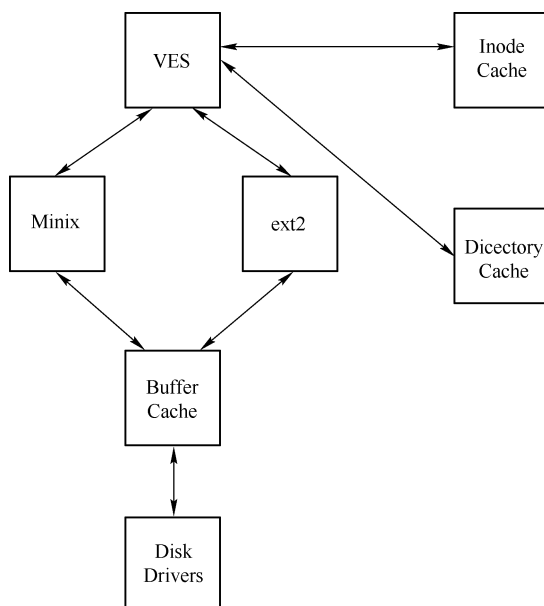


图 7-3 虚拟文件系统的逻辑示意图

文件系统初始化时将其自身注册到 VFS 中，它发生在系统启动和操作系统初始化时。这些实际文件系统可以构造到核心中，也可以设计成可加载模块。文件系统模块可以在系统需要时进行加载，如 VFAT 就被实现成一个核心模块，当 mount VFAT 文件系统时它将被加载。mount 一个基于块设备且包含根文件系统的文件系统时，VFS 必须读取其超块。每个文件系统类型的超块读取例程必须了解文件系统的拓扑结构并将这些信息映射到 VFS 超块结构中。VFS 在系统中保存着一组已安装文件系统的链表及其 VFS 超块。每个 VFS 超块包含一些信息和一个执行特定功能的函数指针。例如，表示一个已安装 ext2 文件系统的超块包含一个指向 ext2 相关 inode 读例程的指针，这个 ext2 inode 读例程像所有文件系统相关读例程一样填充了 VFS inode 中的域。每个 VFS 超块包含此文件系统中第一个 VFS inode 的指针。对于根文件系统此 inode 表示的是“/”目录。这种信息映射方式对 ext2 文件系统非常有效，但是对其他文件系统要稍差一些。

系统中进程访问目录和文件时将使用系统调用遍历系统的 VFS inode。例如，输入 ls 或 cat 命令会引起虚拟文件系统对表示此文件系统的 VFS inode 的搜寻。由于系统中每个文件与目录都使用一个 VFS inode 来表示，所以许多 inode 会被重复访问。这些 inode 被保存在 inode cache 中以加快访问速度。如果某个 inode 不在 inode cache 中，则必须调用一个文件系统相关例程来读取此 inode。对这个 inode 的读取将把它放到 inode cache 中以备下一次访问。不经常使用的 VFS inode 将会从 cache 中移出。

所有 Linux 文件系统都使用一个通用 buffer cache 来缓冲来自底层设备的数据，以便加速对包含此文件系统的物理设备的存取。这个 buffer cache 与文件系统无关并被集成到 Linux 核心分配与读写数据缓存的机制中。让 Linux 文件系统独立于底层介质和设备驱动的

好处很多。所有的块结构设备都将其自身注册到 Linux 核心中并提供基于块的一致性异步接口，像 SCSI 设备这种相对复杂的块设备也是如此。当实际文件系统从底层物理磁盘读取数据时，块设备驱动将从它们所控制的设备中读取物理块。buffer cache 也被集成到了块设备接口中。当文件系统读取数据块时它们将被保存在由所有文件系统和 Linux 核心共享的全局 buffer cache 中，这些 buffer 由其块号和读取设备的设备号来表示。所以若某个数据块被频繁使用则它很可能是从 buffer cache 而不是磁盘中读取出来，后者显然将花费更长的时间。有些设备支持通过预测将下一次可能使用的数据提前读取出来。

VFS 还支持一种目录 cache 以便对经常使用的目录对应的 inode 进行快速查找。我们可以做一个这样的实验，首先对一个最近没有执行过列目录操作的目录进行列目录操作，第一次列目录时可能发现会有较短的停顿但第二次操作时结果会立刻出现。目录 cache 不存储目录本身的 inode，后者应该在 inode cache 中，目录 cache 仅仅保存全目录名及其和 inode 号之间的映射关系。

与 ext2 文件系统相同，VFS 中的每个文件、目录等都只用一个 VFS inode 表示。每个 VFS inode 中的信息通过文件系统相关例程从底层文件系统中得到。VFS inode 仅存在于核心内存并被保存，只要对系统有用它们就会被保存在 VFS inode cache 中。每个 VFS inode 都包含下列域。

1) device

包含此文件或此 VFS inode 代表的任何东西的设备的设备标志符。

2) inode number

文件系统中唯一的 inode 号。在虚拟文件系统中 device 和 inode 号的组合是唯一的。

3) mode

与 ext2 中的相同，表示此 VFS inode 的存取权限。

4) user ids

所有者的标志符。

5) times

VFS inode 创建、修改和写入时间。

6) block size

以字节计算的文件块大小，如 1 024 字节。

7) inode operations

指向一组例程地址的指针。这些例程和文件系统相关且对此 inode 执行操作，如截断此 inode 表示的文件。

8) count

使用此 VFS inode 的系统部件数。一个 count 为 0 的 inode 可以被自由地丢弃或重新使用。

9) lock

用来对某个 VFS inode 加锁，如用于读取文件系统时。

10) dirty

表示这个 VFS inode 是否已经被写过，如果是则底层文件系统需要更新。

11) file system specific information

7.2 注册文件系统

当重新建立 Linux 核心时安装程序会询问是否需要所有可支持的文件系统。核心重建时文件系统启动代码包含了所有编入核心的文件系统的初始化例程。Linux 文件系统可构造成模块，此时它们会只在需要时加载或者使用 `insmod` 来载入。当文件系统模块被加载时，它将向核心注册并在卸载时撤除注册。每个文件系统的初始化例程还将向虚拟文件系统注册，它用一个包含文件系统名称和指向其 VFS 超块读例程指针的 `file_system_type` 结构表示，见图 7-4。

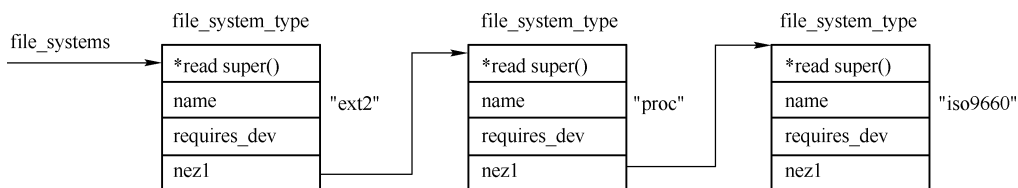


图 7-4 已注册文件系统

每个 `file_system_type` 结构包含下列信息。

1) superblock read routine

此例程在文件系统的实例被安装时由 VFS 调用。

2) file system name

文件系统的名称，如 `ext2`。

3) device needed

文件系统是否需要设备支持。并不是所有的文件系统都需要设备来保存它，例如，`/proc` 文件系统不需要块设备支持。

可以通过查阅 `/proc/filesystems` 找出已注册的文件系统，如：

```
ext2
nodev proc
iso9660
```

7.3 安装文件系统

当超级用户试图安装一个文件系统时，Linux 核心首先使系统调用中的参数有效化。尽管 `mount` 程序会做一些基本的检查，但是它并不知道核心构造时已经支持那些文件系统，同时那些建议的安装点的确存在。看如下的一个 `mount` 命令：

```
$ mount -t iso9660 -o ro /dev/cdrom /mnt/cdrom
```

`mount` 命令将传递 3 个参数给核心：文件系统名、包含文件系统的物理块设备及此新文件系

统要安装到的已存在的目录名。

虚拟文件系统首先必须做的是找到此文件系统。它将通过由链指针 `file_systems` 指向的 `file_system_type` 结构来在所有已知文件系统中搜寻。如果找到一个相匹配的文件系统名，那么它就知道核心支持此文件系统并可得到读取此文件系统超块相关例程的指针。如果找不到，但文件系统使用了可动态加载核心模块，则操作仍可继续。此时核心将请求核心后台进程加载相应的文件系统模块。接下来如果由 `mount` 传递的物理设备还没有安装，则必须找到新文件系统将要安装到的那个目录的 VFS inode。这个 VFS inode 可能在 inode cache 中，也可能在支撑这个安装点所在文件系统的块设备中。一旦找到这个 inode，则将对它进行检查以确定在此目录中是否已经安装了其他类型的文件系统。多个文件系统不能使用相同目录作为安装点。此时 VFS 安装代码必须分配一个 VFS 超块并将安装信息传递到此文件系统的超块读例程中。系统中所有的 VFS 超块都被保存在由 `super_block` 结构构成的 `super_blocks` 数组中，并且对应此安装应有一个这种结构。超块读例程将基于从物理设备中读取的信息来填充这些 VFS 超块域。对于 ext2 文件系统此信息的转化过程十分简便，仅需读取 ext2 超块并填充 VFS 超块。但其他文件系统如 MS-DOS 文件系统就没有那么容易了。无论哪种文件系统，对 VFS 超块的填充意味着文件系统必须从支持它的块设备中读取描述它的所有信息。如果块设备驱动不能从中读取或不包含这种类型的文件，系统则 `mount` 命令会失败。

每个文件系统都用一个 `vfsmount` 结构来描述，如图 7-5 所示，它们被排入由 `vfsmntlist` 指向的链表中。

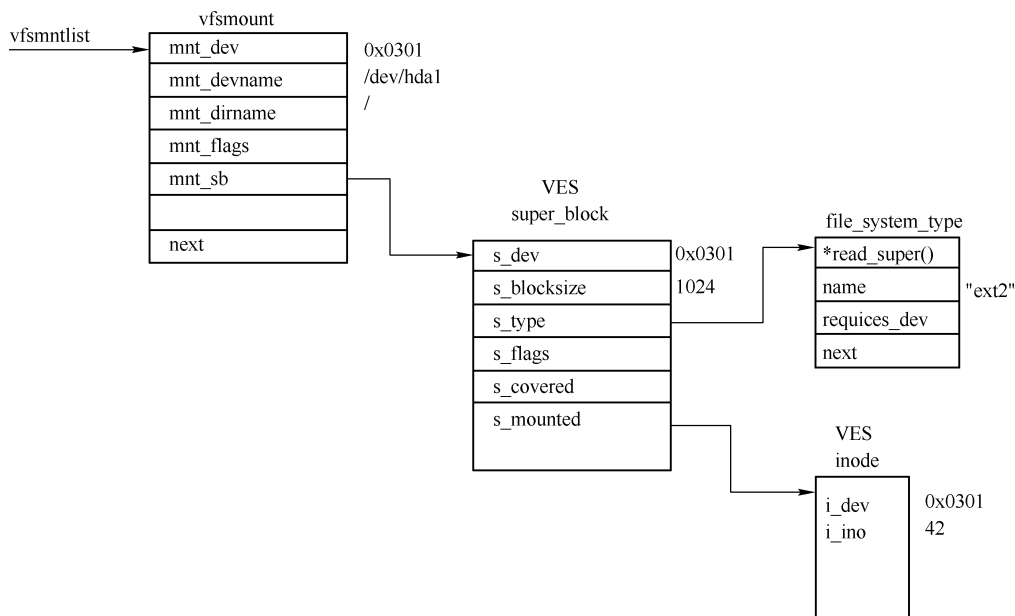


图 7-5 一个已安装的文件系统

另外一个指针 `vfsmnttail` 指向链表的最后一个入口，同时 `mru_vfsmnt` 指针指向最近使用最多的文件系统。每个 `vfsmount` 结构都由以下部分组成：包含此文件系统的块设备的设备号，此文件系统安装的目录及文件，系统安装时分配的 VFS 超块指针。VFS 超块指向这种

类型文件系统和此文件系统根 inode 的 `file_system_type` 结构。一旦此文件系统被加载, 这个 inode 将一直驻留在 VFS inode cache 中。

7.4 在虚拟文件系统中搜寻文件

为了在虚拟文件系统中找到某个文件的 VFS inode, VFS 必须依次解析此文件名字中的间接目录直到找到此 VFS inode。每一次目录查找都包括一个对包含在表示父目录 VFS inode 中的查找函数的调用。由于我们总是让每个文件系统的根可用并且由此系统的 VFS 超块指向它, 所以这是一个可行方案。每次在实际文件系统中寻找 inode 时, 文件系统将在目录 cache 中寻找相应目录。如果在目录 cache 中无相应入口, 则文件系统必须从底层文件系统或 inode cache 中取得此 VFS inode。

7.5 卸载文件系统

如果已安装文件系统中有些文件还在被系统使用, 则不能卸载此文件系统。例如, 有进程使用 `/mnt/cdrom` 或其子目录时将不能卸载此文件系统。如果将要卸载的文件系统中有些文件还在被使用, 则在 VFS inode cache 中有与其对应的 VFS inode。通过在 inode 链表中查找该文件系统占用设备的 inode 来完成此工作。对应此已安装文件系统的 VFS 超块为 `dirty`, 表示它已被修改过所以必须写回到磁盘的文件系统中。一旦写入磁盘, VFS 超块占用的内存将归还到核心的空闲内存池中, 最后对应的 `vfsmount` 结构将从 `vfsmntlist` 中释放。

下面介绍 `bdflush` 核心后台进程。

`bdflush` 是对过多的 `dirty` 缓冲系统提供动态响应的简单核心后台进程, 这些缓冲块中包含必须被写入到硬盘上的数据。它在系统启动时作为一个核心线程运行, 其名称是“`kflushd`”。可以使用 `ps` 命令看到此系统进程。通常情况下此进程一直在睡眠, 直到系统中的 `dirty` 缓冲数目增大到一定数目。当分配与丢弃缓冲时, 系统中 `dirty` 缓冲的数目将进行一个统计, 如果其数目超过某个数值则唤醒 `bdflush` 进程。默认的阈值为 60%, 但是如果系统急需缓冲则任何时刻都可能唤醒 `bdflush`。使用 `update` 命令可以看到和改变这个数值。

7.6 /proc 文件系统

`/proc` 文件系统真正显示了 Linux 虚拟文件系统的能力。事实上它并不存在, 无论 `/proc` 目录还是其子目录和文件都不是真正存在的。但是我们是如何能够执行 `cat/proc/devices` 命令的呢? `/proc` 文件系统像一个真正的文件系统一样会向虚拟文件系统注册。然而当有对 `/proc` 中的文件和目录的请求发生时, VFS 系统将从核心中的数据里临时构造这些文件和目录。例如, 核心的 `/proc/devices` 文件就是从描述其设备的内核数据结构中产生出来的。`/proc` 文件系统提供给用户一个核心内部工作的可读窗口, Linux 核心模块都在 `/proc` 文件系统中创建入口。

7.7 设备特殊文件

和所有的 UNIX 版本一样, Linux 将硬件设备看成特殊的文件, 如 `/dev/null` 表示一个空设备。设备文件不使用文件系统中的任何数据空间, 它只是对设备驱动访问的入口点。ext2 文件系统和 Linux VFS 都将设备文件实现成特殊的 inode 类型。有两种类型的设备文件: 字符与块设备特殊文件。在核心内部设备驱动实现了类似文件的操作过程: 可以对它执行打开、关闭等工作。字符设备允许以字符模式进行 I/O 操作, 而块设备的 I/O 操作需要通过 buffer cache 完成。对一个设备文件发出的 I/O 请求将被传递到相应的设备驱动。通常这种设备文件并不是一个真正的设备驱动, 而只是一个伪设备驱动, 如 SCSI 设备驱动层。设备文件通过表示设备类型的主类型标志符和表示单元或主类型实例的从类型来引用。例如, 在系统中, 第一个 IDE 控制器上的 IDE 硬盘的主设备号为 3, 而其第一个分区的从标志符为 1。

7.8 常见的 Flash 文件系统

文件系统都会被烧录在某一存储设备上。在嵌入式设备上很少使用大容量的 IDE 硬盘作为自己的存储设备, 嵌入式设备往往选用 ROM、闪存 (Flash Memory) 等作为它的主要存储设备。在嵌入式设备上选用哪种文件系统格式是与闪存的特点相关的。

7.8.1 Flash 的特点

(1) 闪存的最小寻址单位是字节 (byte), 而不是磁盘上的扇区 (sector)。这意味着可以从一块闪存的任意偏移 (offset) 读数据, 但并不表明对闪存的写操作也是以字节为单位进行的。

(2) 当一块闪存处在干净的状态时 (被擦写过, 但是还没有写操作发生), 在这块 Flash 上的每一位 (bit) 都是逻辑 1。

(3) 闪存上的每一位 (bit) 可以被写操作置成逻辑 0。但是把逻辑 0 置成逻辑 1 却不能按位 (bit) 来操作, 而只能按擦写块 (erase block) 为单位进行擦写操作。擦写块的大小从 4KB 到 128KB 不等。从上层来看, 擦写所完成的功能就是把擦写块内的每一位都重置 (reset) 成逻辑 1。

(4) 闪存的使用寿命是有限的。具体来说, 闪存的使用寿命是由擦写块的最大可擦写次数来决定的。超过了最大可擦写次数, 这个擦写块就成为坏块 (bad block)。因此为了避免某个擦写块被过度擦写, 以至于它先于其他的擦写块达到最大可擦写次数, 应该在尽量减小影响性能的前提下, 使擦写操作均匀分布在每个擦写块上, 这个过程叫做磨损平衡 (wear leveling)。

目前的 Flash 有两种: NAND Flash 和 NOR Flash。

NOR 型闪存可以直接读取芯片内存储的数据, 因而速度比较快, 但是价格较高。NOR 型芯片的地址线与数据线分开, 所以 NOR 型芯片可以像 SRAM 一样连在数据线上。对

NOR 芯片可以“字”为基本单位进行操作，因此传输效率很高，应用程序可以直接在 Flash 内运行，不必再把代码读到系统 RAM 中运行。它与 SRAM 的最大不同在于，写操作需要经过擦除和写入两个过程。

NAND 型闪存芯片共用地地址线与数据线，内部数据以块为单位进行存储，直接将 NAND 芯片作为启动芯片比较难。NAND 闪存是连续的存储介质，适合放大文件。擦除 NOR 器件时是以 64~128KB 的块进行的，执行一个写入/擦除操作的时间为 5s；擦除 NAND 器件是以 8~32KB 的块进行的，执行相同的操作最多只需要 4ms 时间。NAND 闪存的单元尺寸几乎是 NOR 器件的一半，由于生产过程更为简单，NAND 结构可以在给定的模具尺寸内提供更高的容量，也就相应地降低了价格。NOR Flash 占据了容量为 1~16MB 闪存市场的大部分，而 NAND Flash 只是用在 8~128MB 的产品当中，这也说明 NOR 主要应用在代码存储介质中，NAND 适合于数据存储。寿命（耐用性），在 NAND 闪存中每个块的最大擦写次数是一百万次，而 NOR 的擦写次数是十万次。NAND 存储器除了具有 10:1 的块擦除周期优势，典型的 NAND 块尺寸要比 NOR 器件小 8 倍，每个 NAND 存储器块在给定时间内的删除次数也要少一些。

NAND 的实际应用方式要比 NOR 复杂得多。NOR 可以直接使用，并在上面直接运行代码。而 NAND 需要 I/O 接口，因此使用时需要驱动程序。不过当今流行的操作系统对 NAND Flash 都有支持，Linux 内核也对 NAND Flash 提供了很好的支持。以上 Flash 的特性决定了在嵌入式设备中我们一般会把只读属性的映像文件，如启动引导程序 blob、内核、文件系统文件存放在 NOR Flash 中，而把一些读写类的文件，如用户应用程序等存放在 NAND Flash 中。出于成本的考虑，很多厂家会选用低容量昂贵的 NOR Flash 存储启动引导程序和内核，而把文件系统存放在 NAND Flash 中。

文件系统类型如 dos 和 ext2 都可以安装在嵌入式 Linux 上，只是由于 Flash 的特点及嵌入式环境，使得这些不是特别适合 Flash。如果不进行特别处理，会引起数据丢失、系统崩溃等问题；另外对 Flash 的寿命也没有考虑。目前，Flash 上的文件系统种类很多，有 CRAMFS、JFFS2、YAFFS 等。CRAMFS 是只读文件系统，适合不需要对文件进行修改的场合；JFFS2 是可读写的，适合 NOR Flash；YAFFS 是可读写的，适合 NAND Flash。

将磁盘文件系统（ext2、FAT）运行在闪存上的很自然的方法就是在文件系统和闪存之间提供一个闪存转换层（Flash Translation Layer, FTL），如图 7-6 所示，它的功能是将底层的闪存模拟成一个具有 512B 扇区大小的标准块设备（block device）。对于文件系统来说，就像工作在一个普通的块设备上一样，没有任何的差别。

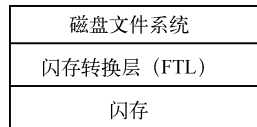


图 7-6 FTL 转换层

一个闪存转换层的最简单的实现是将模拟的块设备一对一地映射到闪存上。举例来说，当上层的文件系统要写一个块设备的扇区时，闪存转换层要进行下面的操作来完成这个写请求：

- （1）将这个扇区所在擦写块的数据读到内存中，放在缓存（buffer）中；
- （2）将缓存中与这个扇区对应的内容用新的内容替换掉；
- （3）对该擦写块执行擦写操作；

(4) 将缓存中的数据写回该擦写块。

这种实现方式的缺点是很明显的：

(1) 效率低，对一个扇区的更新要重写整个擦写块上的数据，造成数据带宽很大的浪费；

(2) 没有提供磨损平衡，那些被频繁更新的数据所在的擦写块将首先变成坏块；

(3) 非常不安全，很容易引起数据的丢失。如果在上面的第(3)步和第(4)步之间发生了突然掉电(power loss)，那么整个擦写块中的数据就全部丢失了。这在突然掉电经常发生的嵌入式系统中是不能接受的。

MTD 中的内核模块 `mtddblock` 就是基于这种机制实现的，同时还进行了一些优化。只有当文件系统的写请求超过了一个擦写块的边界的时候，它才会执行对闪存的擦写、写回操作。

因此，为了解决上面这种实现方式的问题，闪存转换层需要做更多的事情。闪存转换层不能只实现这种一对一的映射，而需要将模拟块设备的扇区存储在闪存的不同位置，并且维持扇区到闪存的映射关系。更进一步，闪存转换层还必须能理解上层文件系统的语义，否则闪存转换层就无法做垃圾回收(Garbage Collection)。这种实现方式最大的问题是效率不高，具体来说，闪存转换层为了能理解上层文件系统的语义，必须对文件系统的每个写请求进行解析，这势必带来写操作性能的下降。另外，要求文件系统下面的一层去理解文件系统的语义，很显然这不是最好的解决方式。我们还有更好的解决问题的方法，就是实现一个特别针对闪存的文件系统，而 JFFS2 就是这样一个文件系统。

7.8.2 JFFS2

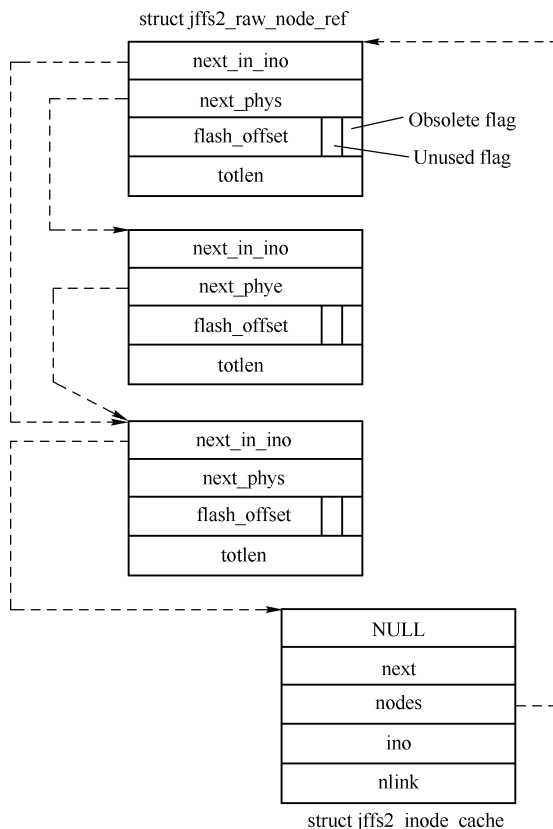
JFFS2 是 RedHat 公司基于 JFFS 开发的闪存文件系统，最初是针对 RedHat 公司的嵌入式产品 eCos 开发的嵌入式文件系统，所以 JFFS2 也可以用在 Linux、uClinux 中。JFFS 文件系统是最早由瑞典 Axis Communications 公司以 Linux 2.0 的内核为嵌入式系统开发的文件系统。JFFS2 是一个可读写的、压缩的、日志型文件系统，并提供了崩溃/掉电安全保护，克服了 JFFS 的一些缺点：使用基于哈希表的日志节点结构，大大加快了对节点的操作速度；支持数据压缩；提供“写平衡”支持；支持多种节点类型；提高了对闪存的利用率，降低了内存的消耗。这些特点使 JFFS2 文件系统成为目前 Flash 设备上最流行的文件系统格式，它的缺点是当文件系统已满或接近满时，JFFS2 的运行会变慢，这主要是因为碎片收集的问题。

下面详细介绍 JFFS2 设计中主要的思想、关键的数据结构和垃圾收集机制。这将为我们实现一个闪存上的文件系统提供很好的启示。JFFS2 是一个日志结构(log-structured)的文件系统，包含数据和元数据(meta-data)的节点在闪存上顺序地存储。

1. JFFS2 节点、擦写块在内存中的表示和操作

JFFS2 维护了几个链表来管理擦写块，根据擦写块上的内容，一个擦写块会在不同的链表上。具体来说，当一个擦写块上都是合法(valid)的节点时，它会在 `clean_list` 上；当一个擦写块包含至少一个过时(obsolete)的节点时，它会在 `dirty_list` 上；当一个擦写块被擦写完毕，并被写入 `cleanmarker` 节点后，它会在 `free_list` 上。

通常情况下, JFFS2 顺序地在擦写块上写入不同的节点, 直到一个擦写块被写满。此时 JFFS2 从 `free_list` 上取下一个擦写块, 继续从擦写块的开头开始写入节点。当 `free_list` 上擦写块的数量逐渐减少到一个预先设定的阈值时, 垃圾回收就被触发了, 为文件系统清理出更多的可用擦写块。为了减少对内存的占用, JFFS2 并没有把 `i` 节点所有的信息都保留在内存中, 而只是把那些在请求到来时不能很快获得的信息保留在内存中。具体来说, 对于在闪存上的每一个 `i` 节点, 在内存里都有一个 `struct jffs2_inode_cache` 与之对应, 这个结构里保存了 `i` 节点号、指向 `i` 节点的连接数, 以及一个指向属于这个 `i` 节点的物理节点链表的指针。所有的 `struct jffs2_inode_cache` 都存储在一个哈希表中。闪存上的每个节点在内存中由一个 `struct jffs2_raw_node_ref` 表示, 见图 7-7。这个结构里保存了此节点的物理偏移、总长度, 以及两个指向 `struct jffs2_raw_node_ref` 的指针。一个指针指向此节点在物理擦写块上的下一个节点, 另一个指针指向属于同一个 `i` 节点的物理节点链表的下一个节点。

图 7-7 `struct jffs2_raw_node_ref` 结构

在闪存上的节点的起始偏移都是 4B 对齐的, 所以 `struct jffs2_inode_cache` 中 `flash_offset` 的最低两位没有被用到。JFFS2 正好利用最低位作为此节点是否过时的标记。

下面举例来说明 JFFS2 是如何使用这些数据结构的。VFS 调用 `iget()` 来得到一个 `i` 节点的信息, 当这个 `i` 节点不在缓存中时, VFS 就会调用 JFFS2 的 `read_inode()` 回调函数来得到 `i`

节点信息。传给 `read_inode()` 的参数是 `i` 节点号, JFFS2 用这个 `i` 节点号从哈希表中查找相应的 `struct jffs2_inode_cache`, 然后利用属于这个 `i` 节点的节点链表从闪存中读入节点信息, 建立映射信息。

JFFS2 的挂载过程分为 4 个阶段, 如下所示。

(1) JFFS2 扫描闪存介质, 检查每个节点 CRC 校验码的合法性, 同时分配 `struct jffs2_inode_cache` 和 `struct jffs2_raw_node_ref`。

(2) 扫描每个 `i` 节点的物理节点链表, 标识出过时的物理节点; 对每一个合法的 `dentry` 节点, 将相应的 `jffs2_inode_cache` 中的 `nlink` 加 1。

(3) 找出 `nlink` 为 0 的 `jffs2_inode_cache`, 释放相应的节点。

(4) 释放在扫描过程中使用的临时信息。

2. JFFS2 垃圾回收机制

当 `free_list` 上的擦写块数太少时, 垃圾回收就会被触发。垃圾回收的主要任务是回收那些已经过时的节点, 但是除此之外它还要考虑磨损平衡的问题。因为如果一味地从 `dirty_list` 上选取擦写块进行垃圾回收, 那么 `dirty_list` 上的擦写块将先于 `clean_list` 上的擦写块被磨损坏。JFFS2 的处理方式是以 99% 的概率从 `dirty_list`、以 1% 的概率从 `clean_list` 上取一个擦写块下来。由此可以看出, JFFS2 的设计思想是偏向于性能, 同时兼顾磨损平衡。对这个块上每一个没有过时的节点执行以下相同的操作:

(1) 找出这个节点所属的 `i` 节点号。

(2) 调用 `iget()`, 建立这个 `i` 节点的文件映射表。

(3) 找出这个节点上没有过时的数据内容, 并且如果合法的数据太少, JFFS2 还会合并相邻的节点。

(4) 将数据读入到缓存中, 然后将它复制到新的擦写块上。

(5) 将回收的节点置为过时。

当擦写块上所有的节点都被置为过时后, 就可以擦写这个擦写块, 回收使用它了。

7.8.3 YAFFS 文件系统

YAFFS/YAFFS2 是一种与 JFFSx 类似的闪存文件系统, 它是专为嵌入式系统使用 NAND 型闪存而设计的一种日志型文件系统。与 JFFS2 相比它减少了一些功能, 所以速度更快, 而且对内存的占用较小。此外, YAFFS 自带 NAND 芯片的驱动, 并且为嵌入式系统提供了直接访问文件系统的 API, 用户可以不使用 Linux 中的 MTD 与 VFS, 直接对文件系统进行操作。YAFFS2 支持大页面的 NAND 设备, 并且对大页面的 NAND 设备进行了优化。JFFS2 在 NAND 闪存上的表现并不稳定, 而更适合于 NOR 闪存, 所以相对于大容量的 NAND 闪存, YAFFS 是更好的选择。

7.9 根文件系统

每台机器都有根文件系统, 它包含系统引导并可使其他文件系统得以 `mount` 所必要的文件, 根文件系统应该有单用户状态所必需的足够的内容, 还应该包括修复损坏系统、恢复

备份等的工具。根文件系统一般应该比较小，因为包括严格的文件和一个小的不经常改变的文件系统不容易损坏。损坏的根文件系统通常意味着除非用特定的方法（如从软盘），否则系统无法引导。

根目录一般不含任何文件，除了可能的标准的系统引导映像，通常叫/vmlinuz。所有其他文件都在根文件系统的子目录中。

- /bin: 引导启动所需的命令或普通用户可能用的命令(可能在引导启动后)。
- /sbin: 类似/bin, 但不给普通用户使用, 虽然如果必要且允许时可以使用。
- /etc: 特定机器的配置文件。
- /root: root 用户的目录。
- /lib: 根文件系统上的程序所需的共享库。
- /lib/modules: 核心可加载模块, 特别是恢复损坏系统时引导所需的（如网络 and 文件系统驱动）。
- /dev: 设备文件。
- /tmp: 临时文件。引导启动后运行的程序应该使用/var/tmp, 而不是/tmp, 因为前者可能在一个拥有更多空间的磁盘上。
- /boot: 引导加载器（bootstrap loader）使用的文件, 如 LILO。核心映像也经常在这里, 而不是在根目录下。如果有许多核心映像, 这个目录可能变得很大, 这时可能使用单独的文件系统更好。
- /mnt: 系统管理员临时 mount 的安装点。程序并不自动支持安装到/mnt。/mnt 可以分子目录（例如, /mnt/dosa 可能是使用 MSDOS 文件系统的软驱, 而/mnt/exta 可能是使用 ext2 文件系统的软驱）。
- /proc、/usr、/var、/home: 其他文件系统的安装点。

以下详细介绍每个目录。

1. /etc 目录

/etc 目录包含很多文件。许多网络配置文件也在/etc 中。

- /etc/rc、/etc/rc.d、/etc/rc*.d: 启动或改变运行级时运行的 scripts 或 scripts 目录。
- /etc/passwd: 用户数据库, 其中的域给出了用户名、真实姓名、home 目录、加密的口令和用户的其他信息。
- /etc/fstab: 启动时 mount -a 命令（在/etc/rc 或等效的启动文件中）自动 mount 的文件系统列表。在 Linux 下, 也包括用 swapon -a 启用的 swap 区的信息。
- /etc/group: 类似/etc/passwd, 但说明的不是用户而是组。
- /etc/inittab: init 的配置文件。
- /etc/issue: getty 在登录提示符前的输出信息。通常包括系统的一段短说明或欢迎信息, 内容由系统管理员确定。
- /etc/magic: file 的配置文件。包含不同文件格式的说明, file 基于它猜测文件类型。
- /etc/motd: Message Of The Day, 成功登录后自动输出。内容由系统管理员确定。经常用于通告信息, 如计划关机时间的警告。
- /etc/mtab: 当前安装的文件系统列表。由 scripts 初始化, 并由 mount 命令自动更

新。在需要一个当前安装的文件系统的列表时使用，例如 `df` 命令。

- `/etc/shadow`: 在安装了影子口令软件的系统上的影子口令文件。影子口令文件将 `/etc/passwd` 文件中的加密口令移动到 `/etc/shadow` 中，而后者只对 `root` 可读。这使破译口令更困难。
- `/etc/login.defs`: `login` 命令的配置文件。
- `/etc/printcap`: 类似 `/etc/termcap`，但针对打印机，语法不同。
- `/etc/profile`、`/etc/csh.login`、`/etc/csh.cshrc`: 登录或启动时 Bourne 或 C shells 执行的文件。这允许系统管理员为所有用户建立全局默认环境。
- `/etc/securetty`: 确认安全终端，即哪个终端允许 `root` 登录。一般只列出虚拟控制台，这样就不可能（至少很困难）通过 MODEM 或网络闯入系统并得到超级用户特权。
- `/etc/shells`: 列出可信任的 shell。`chsh` 命令允许用户在本文件指定范围内改变登录 shell。提供一台机器，FTP 服务的服务进程 `ftpd` 检查用户 shell 是否列在 `/etc/shells` 文件中，如果不是将不允许该用户登录。
- `/etc/termcap`: 终端性能数据库。说明不同的终端用什么“转义序列”控制。写程序时不直接输出转义序列（这样只能工作于特定品牌的终端），而是从 `/etc/termcap` 中查找要做工作的正确序列。这样，多数的程序可以在多数终端上运行。

2. `/dev` 目录

`/dev` 目录包括所有设备的设备文件。设备文件用特定的约定命名。

3. `/usr` 文件系统

`/usr` 文件系统经常很大，因为所有程序都安装在这里。`/usr` 中的所有文件一般来自 Linux distribution；本地安装的程序和其他文件在 `/usr/local` 下。这样可能在升级新版系统或新 distribution 时无须重新安装全部程序。

- `/usr/X11R6`: X Window 系统的所有文件。为简化 X 的开发和安装，X 的文件没有集成到系统中。X 自己在 `/usr/X11R6` 下类似于 `/usr`。
- `/usr/X386`: 类似 `/usr/X11R6`，但是给 X11 Release 5 的。
- `/usr/bin`: 几乎所有的用户命令。有些命令在 `/bin` 或 `/usr/local/bin` 中。
- `/usr/sbin`: 根文件系统不必要的系统管理命令，如多数服务程序。
- `/usr/man`、`/usr/info`、`/usr/doc`: 手册页、GNU 信息文档和各种其他文档文件。
- `/usr/include`: C 编程语言的头文件。为了一致性，它实际上应该在 `/usr/lib` 下，但传统上支持这个名字。
- `/usr/lib`: 程序或子系统的不变的数据文件，包括一些 site-wide 配置文件。名字 `lib` 来源于库 (library)；编程的原始库存放在 `/usr/lib` 中。
- `/usr/local`: 本地安装的软件和其他文件放在这里。

4. `/var` 文件系统

`/var` 包括系统运行时要改变的数据。每个系统都是特定的，即不通过网络与其他计算机共享。

- `/var/catman`: 当要求格式化时的 man 页的 cache。man 页的源文件一般存放在

/usr/man/man*中；有些 man 页可能有预格式化的版本，存放在/usr/man/cat*中。而其他的 man 页在第一次看时需要格式化，格式化完的版本存放在/var/man 中，这样其他人再看相同的页时就无须等待格式化了（/var/catman 经常被清除，就像清除临时目录一样）。

- /var/lib：系统正常运行时要改变的文件。
- /var/local：/usr/local 中安装的程序的可变数据（即系统管理员安装的程序）。注意，如果有必要，即使本地安装的程序也会使用其他的/var 目录，如/var/lock。
- /var/lock：锁定文件。许多程序遵循在/var/lock 中产生一个锁定文件的约定，以支持它们正在使用某个特定的设备或文件。其他程序注意到这个锁定文件，将不会试图使用这个设备或文件。
- /var/log：各种程序的 Log 文件，特别是 login（/var/log/wtmp log 所有到系统的登录和注销）和 syslog（/var/log/messages 中存储所有核心和系统程序信息）。/var/log 里的文件经常不确定地增长，应该定期清除。
- /var/run：保存到下次引导前有效的关于系统的信息文件。例如， /var/run/utmp 包含当前登录的用户的信息。
- /var/spool：mail、news、打印队列和其他队列工作的目录。每个不同的 spool 在 /var/spool 下有自己的子目录，例如，用户的邮箱在/var/spool/mail 中。
- /var/tmp：比/tmp 允许的大或需要存在较长时间的临时文件（虽然系统管理员可能不允许/var/tmp 有很旧的文件）。

5. /proc 文件系统

/proc 文件系统是一个假的文件系统，它不存在于某个磁盘上，而是由核心在内存中产生，用于提供关于系统的信息（originally about processes, hence the name）。下面说明一些最重要的文件和目录。

- /proc/1：关于进程 1 的信息目录。每个进程在/proc 下都有一个名为其进程号的目录。
- /proc/cpuinfo：处理器信息，如类型、制造商、型号和性能。
- /proc/devices：当前运行的核心配置的设备驱动列表。
- /proc/dma：显示当前使用的 DMA 通道。
- /proc/filesystems：核心配置的文件系统。
- /proc/interrupts：显示使用的中断。
- /proc/kcore：系统物理内存映像。与物理内存大小完全相等，但不实际占用这么多内存（注意：除非把它复制到什么地方，否则/proc 下没有任何东西占用任何磁盘空间）。
- /proc/kmsg：核心输出的消息。也被送到 syslog。
- /proc/ksyms：核心符号表。
- /proc/loadavg：系统的“平均负载”。3 个指示器指出系统当前的工作量。
- /proc/meminfo：存储器使用信息，包括物理内存和 swap。
- /proc/modules：当前加载了哪些核心模块。

- `/proc/net`: 网络协议状态信息。
- `/proc/self`: 到查看`/proc` 的程序的进程目录的符号连接。当有两个进程查看`/proc` 时, 是不同的连接。这主要便于程序得到它自己的进程目录。
- `/proc/stat`: 系统的不同状态。
- `/proc/uptime`: 系统启动的时间长度。
- `/proc/version`: 核心版本。

第 8 章

将设备联网——嵌入式 Web Server 的实现

基于网络的体系结构是 Web 工作的基本环境，Web 实际处于 TCP/IP 模型的应用层，是一种网络协议的高层应用。从浏览器提交的请求通过 Web 服务器传给应用程序服务器，由它调用相关的网页应用程序进行处理，处理的结果——网页交给 Web 服务器，Web 服务器把这个网页作为对请求的应答发送给浏览器，如图 8-1 所示。

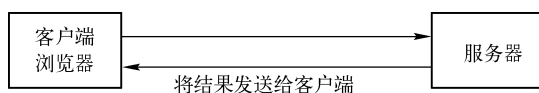


图 8-1 IE 和服务器之间的交互

至于对提交的信息如何处理，则交由网站的开发人员编写相关的网页应用程序来决定反馈到浏览器的内容；如果需要对数据库进行访问，开发人员还可以利用应用程序服务器所提供的接口对其进行操作。如前所述，网页内容的动态发布是 Web 应用程序的主要实现方法，通常这种方法与服务器提供的 WWW 服务技术密不可分。一般来说，在 Internet 服务器上可以通过多种途径实现动态内容的发布，最常见的技术包括 JSP、CGI、ISAPI 和 ASP 等。

了解了 B/S 构架也就理解了网页应用程序的原理。常见的计数器、留言板、聊天室和 BBS 论坛等，都是网页应用程序，只不过这些应用相对比较简单，它的主要应用集中在对数据库的访问上。

8.1 Web 基础知识

8.1.1 HTTP 协议

我们在浏览器的地址栏里输入的网站地址叫 URL (Uniform Resource Locator, 统一资源定位符)。就像每家每户都有一个门牌地址一样, 每个网页也都有一个 Internet 地址。当在浏览器的地址栏中输入一个 URL 或是单击一个超级链接时, URL 就确定了要浏览的地址。浏览器通过超文本传输协议 (HTTP), 将 Web 服务器上站点的网页代码提取出来, 并翻译成漂亮的网页。因此, 在我们认识 HTTP 之前, 有必要先弄清楚 URL 的组成。

例如, `http://www.armv.cn/shop/index.htm`, 它的含义如下。

- (1) `http://`: 代表超文本传输协议, 通知 `armv.cn` 服务器显示 Web 页, 通常不用输入。
- (2) `www`: 代表一个 Web (万维网) 服务器。
- (3) `armv.cn/`: 这是装有网页的服务器的域名, 或站点服务器的名称。
- (4) `shop/`: 为该服务器上的子目录, 类似于文件夹。
- (5) `index.htm`: 是文件夹中的一个 HTML 文件 (网页)。

HTTP 协议是基于请求/响应模式的 (相当于客户端/服务器)。一个客户端与服务器建立连接后, 发送一个请求给服务器, 请求的格式为: 统一资源定位符 (URL)、协议版本号, 后面是 MIME 信息, 包括请求修饰符、客户端信息和可能的内容。服务器接到请求后, 给予相应的响应信息, 其格式为一个状态行, 包括信息的协议版本号、一个成功或错误的代码, 后面是 MIME 信息, 包括服务器信息、实体信息和可能的内容。

8.1.2 HTTP 请求

HTTP 请求由 3 个部分构成, 分别是方法-URI-协议/版本、请求头、请求正文。下面是一个 HTTP 请求的例子:

```
GET /servlet/default.jsp HTTP/1.1
Accept: text/plain; text/html
Accept-Language: en-gb
Connection: Keep-Alive
Host: localhost
Referer: http://localhost/ch8/SendDetails.htm
User-Agent: Mozilla/4.0 (compatible; MSIE 4.01; Windows 98)
Content-Length: 33
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate

userName=JavaJava&userID=javaID
```

请求的第一行是“方法-URI-协议/版本”, 其中 GET 就是请求方法, `/servlet/default.jsp` 表示 URI, `HTTP/1.1` 是协议和协议的版本。根据 HTTP 标准, HTTP 请求可以使用多种请求

方法。例如，HTTP 1.1 支持 7 种请求方法：GET、POST、HEAD、OPTIONS、PUT、DELETE 和 TRACE。在 Internet 应用中，最常用的请求方法是 GET 和 POST。

URI 完整地指定了要访问的网络资源，通常认为它相对于服务器的根目录而言，因此总是以 “/” 开头。URL 实际上是 URI 的一种类型。最后，协议版本声明了通信过程中使用的 HTTP 协议的版本。

请求头包含许多有关客户端环境和请求正文的有用信息。例如，请求头可以声明浏览器所用的语言、请求正文的长度，等等，它们之间用一个回车换行符号（CRLF）分隔。

请求头和请求正文之间是一个空行（只有 CRLF 符号的行），这个行非常重要，它表示请求头已经结束，接下来的是请求的正文。一些介绍 Internet 编程的书籍把这个 CRLF 视为 HTTP 请求的第 4 个组成部分。

在前面的 HTTP 请求中，请求的正文只有一行内容。当然，在实际应用中，HTTP 请求的正文可以包含更多内容。

8.1.3 HTTP 应答

和 HTTP 请求相似，HTTP 应答也由 3 个部分构成，分别是协议-状态代码-描述、应答头、应答正文。下面是一个 HTTP 应答的例子：

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/4.0
Date: Mon, 3 Jan 1998 13:13:33 GMT
Content-Type: text/html
Last-Modified: Mon, 11 Jan 1998 13:23:42 GMT
Content-Length: 112

<html>
<head>
<title>HTTP 应答示例</title></head><body>
Hello HTTP!
</body>
</html>
```

HTTP 应答的第一行类似于 HTTP 请求的第一行，它表示通信所用的协议是 HTTP 1.1，服务器已经成功地处理了客户端发出的请求（200 表示成功），一切顺利。

应答头也和请求头一样包含许多有用的信息，例如，服务器类型、日期时间、内容类型和长度等。应答的正文就是服务器返回的 HTML 页面。应答头和正文之间也用 CRLF 分隔。

许多 HTTP 通信是由一个用户代理初始化的，并且包括一个申请资源服务器上资源的请求，如图 8-2 所示。最简单的情况可能是在用户代理和服务器之间通过一个单独的连接来完成。在 Internet 上，HTTP 通信通常发生在 TCP/IP 连接之上，默认端口是 TCP80，其他的端口也是可用的。但这并不预示着 HTTP 协议在 Internet 或其他网络的其他协议之上才能完成。HTTP 只是预示着一个可靠的传输。

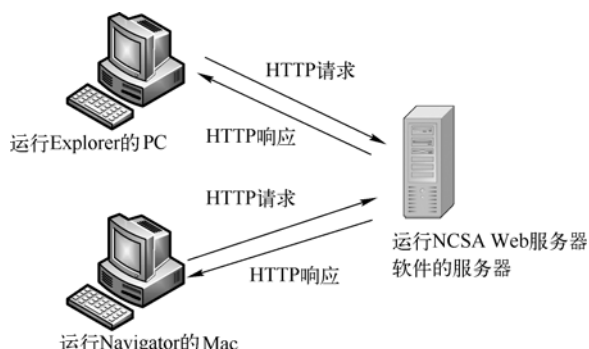


图 8-2 HTTP 通信原理

这个过程就好像我们打电话订货一样，我们可以打电话给商家，告诉对方我们需要什么规格的商品，然后商家再告诉我们什么商品有货、什么商品缺货。这些都是通过电话线用电话联系的（HTTP 是通过 TCP/IP），当然也可以通过传真，只要商家那边也有传真即可。

在 WWW 中，“客户端”与“服务器”是一个相对的概念，只存在于一个特定的连接期间，即在某个连接中的客户端在另一个连接中可能作为服务器。基于 HTTP 协议的客户端/服务器模式的信息交换过程分为 4 部分：建立连接、发送请求信息、发送响应信息、关闭连接。这就好比上面的例子——我们打电话订货的全过程。

“发送服务请求”是什么意思呢？答案很明确，是客户端想要得到某个服务（如想浏览网页）而向服务器发送的请求。服务器在得到请求之后，就会将请求的结果反馈给发送请求的客户端。这样就构成了一个完整的流程。服务器不知疲倦地工作，不停地响应来自于任何地方的不同服务请求，在权限允许的情况下将数据源源不断地发送出去；再深入一点，客户端与服务器是如何连接的呢？是通过任何可能的链路连接的，包括卫星、微波、光纤，对我们来说双绞线、电话线路是最一般的选择。从这个方面可以知道，Web 只是应用，对介质没有要求。

8.2 面向电子商务的 B/S 结构

Browser/Server（浏览器/服务器）结构模式简称 B/S 模式，如图 8-3 和图 8-4 所示。

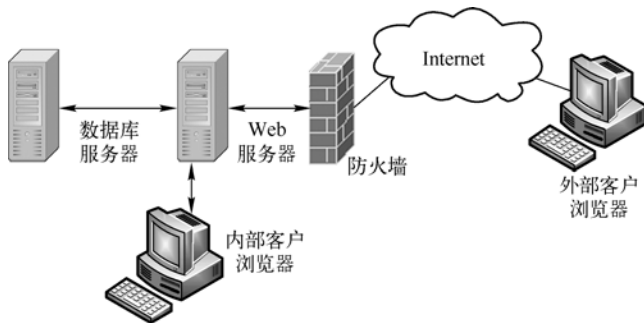


图 8-3 商务网站的 B/S 结构模式

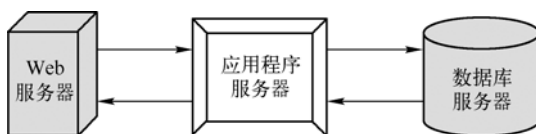


图 8-4 商务网站内部层次结构

在实际的使用过程中，我们可以知道：

- 浏览器——通常是 IE，或运行 IE 等浏览器的计算机；
- 服务器——通常是保存网页的远程服务器。

8.3 Web Server 制作网页

HTML 文件的开头和结尾是由<html>和</html>标记的。所有的 HTML 文件都可以分为两个部分：标题和正文。每一部分都用特定的标记标出：在 HTML 语言中规定，用<head>和</head>标记标题部分，用<body>和</body>标记正文部分。

下面举例说明：

```
<html>
<head>
<title>我的第一个网页</title>
</head>
<body>
<p><font name="隶书" size="7">FrontPage 使用 7 号隶书字体显示的网页</font></p>
</body>
</html>
```

(1) 我们在 IE 中看到的所有内容实际上都是先经过 IE 的处理后才显示出来的。

(2) 程序中<html>与</html>是一对标记，标志着该 HTML 文件的开始及结束。也就是说，一个 HTML 文件中在这两个标记之外的内容，浏览器会（如 IE）统统忽略不去处理（脚本代码除外，后面会讲到）。

(3) <head>与</head>也是一对标记，这两个标记之间的内容是定义整个网页属性的。例如，下面的“<title>我的第一个网页</title>”，就是将网页的标题设置为“我的第一个网页”。

(4) <body>与</body>之间的内容是在浏览器工作区中显示的内容，所谓“工作区”在 IE 中就是工具条下方、状态栏上方的区域。

(5) “<p>FrontPage 使用 7 号隶书字体显示的网页</p>”，这行代码是要求浏览器使用隶书 7 号字体在工作区中显示“FrontPage 使用 7 号隶书字体显示的网页”这句话。标记<p>是换行回车；是指定后续的内容以 7 号隶书字体显示；是结束这种强制显示。

(6) HTML 语言最终被浏览器（如 Internet Explorer）解释执行，显示在浏览器中。

(7) 浏览器对 HTML 语言的解释是按照代码从上至下的顺序执行的。

(8) 任何一种浏览器都具有对标准 HTML 语言进行执行的功能,但细微之处略有差别。

(9) HTML 语言是一种标记语言,使用文本格式实现输出的格式化。

表 8-1 是 HTML 语言的标记语法格式及说明。

表 8-1 HTML 语言的标记语法格式及说明

标记语法格式	说 明
<html>...</html>	表示文件类型为 HTML 文档
<head>...</head>	设置文档描述及其他不在 Web 网页上显示的信息
<body>...</body>	HTML 文档的主体(页面的实际内容)
<title>...</title>	在标题栏中显示的题目(放在<head>和</head>之间)
	设置字体大小,从 1 到 7
	设置字体的颜色,使用名字或十六进制值
	创建一个超链接
 	创建一个自动发送电子邮件的链接
<p>	创建一个新的段落
	添加一个图像
<table>...</table>	创建一个表格
<tr>...</tr>	开始表格中的每一行
<td>...</td>	开始一行中的每一格
<th>...</th>	设置表格头——一个通常使用黑体居中文字的格子

再看一个例子:

```
<html>
<head>
<title>网页 2——红色字显示的网页</title>
</head>
<body>
<font color="#FF0000">红色字显示的网页</font>
</body>
</html>
```

8.4 CGI 工作原理

CGI 即公共网关接口(Common Gateway Interface),是在 Web 服务器上定义 Web 客户端请求与应答的一种方法。客户端向服务器的请求只要属于 CGI 范围,就启动 Web 服务器的一个 CGI(网关)程序。它的任务是把客户端的请求从网关的环境变量(见 8.4.1 节)中取出,并进行相应的加工处理。由 CGI 程序决定如何对客户端的请求做出应答。另外,CGI 程序定义标准的方法为服务器及客户端标准的请求与响应信息。因此,CGI 是 Web 服务器不可缺少的组成部分。

8.4.1 环境变量

CGI 的标准规定服务器必须通过其环境变量来为 CGI 的程序传递信息内容。

通常 HTTPD 服务器的 cgi-bin 目录中用 test-env 的小程序来验证 CGI 返回的环境变量。

例如：

```
SERVER_NAME=lab.fjinfo.sti.ac.cn
HTTP_CONNECTION=Keep-Alive
REMOTE_ADDR=168.160.135.21
HTTP_HOST=lab
REQUEST_METHOD=GET
GATEWAY_INTERFACE=CGI/1.1
QUERY_STRING=
REMOTE_USER=chencx
SERVER_SOFTWARE=NCSA/1.5
SERVER_PROTOCOL=HTTP/1.0
REMOTE_HOST=chencx.fjinfo.sti.ac.cn
SERVER_PORT=80
DOCUMENT_ROOT=/usr/local/etc/httpd/htdocs
HTTP_USER_AGENT=Mozilla/3.0 (Win95; I)
HTTP_ACCEPT=image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
SCRIPT_NAME=/cgi-bin/test-env
SERVER_ADMIN=httpd@lab.fjinfo.sti.ac.cn
AUTH_TYPE=Basic
```

对这些变量给出如下解释：

- **SERVER_SOFTWARE**：是客户端请求的 HTTP 服务器的软件名称及版本。
- **SERVER_NAME**：是 HTTP 服务器的主机域名。
- **GATEWAY_INTERFACE**：是指该服务器的 CGI 版本。
- **SERVER_PROTOCOL**：用于发送请求的协议名称和版本，目前只有 HTTP 协议支持 CGI 的标准。
- **SERVER_PORT**：是用于接收客户端请求的服务器端口号，常用的是 80。
- **REMOTE_HOST**：是指用于发送请求的客户端的主机域名。只有当客户端主机具有指定的域名时，才有该值。
- **REMOTE_ADDR**：是指发送请求的客户端在 HTTP 主机上返回的 IP 地址。
- **REMOTE_USER**：是指远程授权用户的用户名。
- **AUTH_TYPE**：用于保护客户端的授权类型。
- **REQUEST_METHOD**：是客户端用于发送请求的方法。
- **HTTP_ACCEPT**：是客户端支持的 MIME 类型清单，各个类型间用逗号分隔。星号为通配符，它表示可接收任意类型的数据。不过目前许多 HTTP 服务器尚未使用这个变量。
- **PATH_INFO**：是接在 CGI 脚本之后的其他路径名信息。

- QUERY_STRING: 接收来自客户端浏览器的数据。
- CONTENT_TYPE: 用于发送信息的 MIME 类型。目前常用的是 application/x-www-form-urlencoded。
- CONTENT_LENGTH: 是指当客户端使用 POST 方法发送请求时, 计算从客户端浏览器发送的字节数。

8.4.2 CGI 标题和 GET/POST

1. CGI 标题

HTTP 服务器通过环境变量与 CGI 程序进行通信。而 CGI 程序通过标准输出将数据以 HTML 形式送到服务器、再到客户端。CGI 的输出程序必须使用三个标题行, 如下所示。

第一行, Content-type: 设定输出数据的 MIME 输出类型。常用的是 Content-type: plain/text 或者 Content-type: jpeg/gif。

第二行, Location: 是指输出的文档名称。

第三行, Status: 是 HTTP 的状态码。

例如, <http://www.w3.org/hypertext/WWW/Protocols/HTTP/HTRESP.html>。

注意, 这三行必须单独各占一行, 并且每一行之后必须有一个空行。只有这样服务器才能解释 CGI 产生的文档结果。

2. POST/GET 方法

HTTP-GET 和 HTTP-POST 是使用 HTTP 的标准协议动词, 用于编码和传送变量名/变量值对参数, 并且使用相关的请求语义。每个 HTTP-GET 和 HTTP-POST 都由一系列 HTTP 请求头组成, 这些请求头定义了客户端从服务器请求了什么, 而响应则由一系列 HTTP 应答头和应答数据组成, 如果请求成功则返回应答。

HTTP-GET 以使用 MIME 类型 application/x-www-form-urlencoded 的 urlencoded 文本的格式传递参数。urlencoding 是一种字符编码, 保证被传送的参数由遵循规范的文本组成, 例如, 一个空格的编码是 "%20"。附加参数还能被认为是一个查询字符串。

与 HTTP-GET 类似, HTTP-POST 参数也是被 URL 编码的。然而, 变量名/变量值不作为 URL 的一部分被传送, 而是放在实际的 HTTP 请求消息内部被传送。

1) 使用 POST 与使用 GET 的区别

在 Form 里, 可以使用 POST 也可以使用 GET, 它们都是 method 的合法取值。但是, POST 和 GET 方法在使用上至少有两点不同:

(1) GET 方法通过 URL 请求来传递用户的输入, POST 方法通过另外的形式。

(2) GET 方式的提交需要用 Request.QueryString 来取得变量的值, 而 POST 方式的提交必须通过 Request.Form 来访问提交的内容。

仔细研究下面的代码, 可以通过运行来感受一下:

```
<!--两个 Form 只有 Method 属性不同-->  
<FORM ACTION="getpost.asp" METHOD="get">  
<INPUT TYPE="text" NAME="Text" VALUE="Hello World"></INPUT>  
<INPUT TYPE="submit" VALUE="Method=Get"></INPUT>
```

```

</FORM>
<BR>
<FORM ACTION="getpost.asp" METHOD="post">
<INPUT TYPE="text" NAME="Text" VALUE="Hello World"></INPUT>
<INPUT TYPE="submit" VALUE="Method=Post"></INPUT>
</FORM>
<BR>
<BR>

<% If Request.QueryString("Text") <> " " Then %>
通过 GET 方法传递来的字符串是: "<B><%=Request.QueryString("Text") %></B>"<BR>
<% End If %>

<% If Request.Form("Text") <> "" Then %>
通过 POST 方法传递来的字符串是: "<B><%=Request.Form("Text") %></B>"<BR>
<% End If %>

```

说明如下:

把上面的代码保存为 `getpost.asp`, 然后运行。首先测试 POST 方法, 这时浏览器的 URL 并没有什么变化, 返回的结果是:

通过 Post 方法传递来的字符串是: "Hello World"

然后测试用 GET 方法提交, 请注意, 浏览器的 URL 变成了 `http://localhost/general/form/getpost.asp?Text=Hello+World`, 而返回的结果是:

通过 GET 方法传递来的字符串是: "Hello World"

最后再通过 POST 方法提交, 浏览器的 URL 还是 `http://localhost/general/form/getpost.asp?Text=Hello+World`, 而返回的结果变成:

通过 GET 方法传递来的字符串是: "Hello World"
通过 POST 方法传递来的字符串是: "Hello World"

通过 GET 方法提交数据可能会带来安全性的问题。例如, 一个登录页面, 当通过 GET 方法提交数据时, 用户名和密码将出现在 URL 上。如果:

- (1) 登录页面可以被浏览器缓存;
- (2) 其他人可以访问客户的这台机器。

那么, 别人即可从浏览器的历史记录中读取到此客户的账号和密码。所以, 在某些情况下, GET 方法会带来严重的安全性问题。

我们将在后面的实验中把数据传送到服务器的 CGI 脚本中, 即客户端浏览器的信息就是通过 POST 和 GET 两种方式传至服务器的。

2) POST 方法

当客户端的 HTML 的表格设计采用 POST 时, CGI 中的环境变量 `REQUEST_METHOD` 的返回值是 POST。而 `CONTENT_LENGTH` 计算出客户端传送的信息的数据量 (以字节为

单位)。CGI 程序就根据 `CONTENT_LENGTH` 计算出 HTTP 何时开始停止阅读从客户端传来的信息。用 POST 方式发送的信息只有 `application/x-www-form-urlencoded` 一种 MIME 类型数据,并把此类型记录在 `CONTENT_TYPE` 中。客户端无论是以 POST 还是 GET 方式向服务器传送信息,都必须加以编码才能将信息数据传至服务器中。

因此,服务器收到来自客户端的信息数据时,必须对 URL 之后的数据进行解码(客户端浏览器编码的逆过程),使其数据信息恢复原貌。

该过程如下:

- (1) 把原来用 & 连在一起的 HTML 表格变量与其值分开;
- (2) 把原来用 = 连在一起的 HTML 表格变量与其值分开;
- (3) 把各变量和数据中包含十六进制的数转换成 ASCII 码的等价表示;
- (4) 把各变量和数据中的所有 + 替换成空格。

依次把编码的信息数据还原成客户端输入时的原来面貌,此时 CGI 程序就可以对这些数据进行一系列的加工和处理了。

3) GET 方法

GET 方法把客户端的表格数据加到 URL 的末尾,传送至服务器。故必须对其传送的数据进行专门的编码。大家知道,URL 的标准中保留了若干特殊的字符,如 &、#、?、=、空格及任意不可打印的 ASCII 控制字符。这些对我们的 CGI 来讲具有重要的作用。如用 + 来代替原来的空格字符,保留的字符用 % 后接相应的十六进制数的 ASCII 码值替代。大多数的客户端浏览器均可对其进行正确的编码。但是值得注意的是,应防止一些非法的用户通过 CGI 的脚本造成对系统安全的威胁(如 sendmail)。

基于 GET 方法的 CGI 程序与 POST 程序的运行原理基本相同,只是服务器发送的数据采用的方式稍微有些差异。它不是把客户端浏览器上传到服务器的数据发送到 CGI 程序的标准输入,而是将这些数据经编码后由 `QUERY_STRING` 环境变量传输,服务器上的 CGI 程序只要识别 `QUERY_STRING` 即可。另外,环境变量 `REQUEST_METHOD` 的默认值为 GET。

8.4.3 CGI 程序的开发

CGI 程序的工作一般就是接收表单数据,根据应用需求进行数据处理,最后根据处理结果生成新的页面返回给浏览器。表单数据通常以 POST 方法提交给服务器,由 CGI 程序获得,程序根据元素名称/值中的元素名字来区分数据。完成数据处理后,再读取相应的模板文件,根据注释标记将对应的数据填充到 HTML 文本中,生成最后的页面返回给浏览器。

程序执行的一般逻辑为:

- (1) 进行安全性检查,确定是否允许运行脚本;
- (2) 处理用户提交的数据,根据元素名称/值中的元素名字来区分数据,然后根据应用需求进行数据处理;
- (3) 用处理结果填充表单,根据注释标记将对应的数据填充到 HTML 文本中,生成最后的页面并返回给浏览器。

CGI 可以使用多种编程语言来实现, 包括 C、C++、PERL 等。但在嵌入式设备的开发中, 通常不会采用 PERL 等解释性语言, 因为这种语言还需要有解释执行的支撑模块, 会占用存储空间和内存, 最常用的方法是用 C 来编写。

8.4.4 几种常用数据库接口

CGI 程序可以完成与一些大型数据库的接口。通过标准的客户端浏览器达到对大型数据库的检索或产生动态的页面。国际上各大公司对此都非常重视, 都相应开发出一系列产品, 使二者间的接口更容易。例如, 微软公司的 Information Server 与 MS SQL 接口, 还有 Sybase、Informix、Oracle 等。读者也可以编写一些 CGI, 使浏览器能对 Foxpro 数据库进行访问。

8.4.5 几种常用 CGI 及其 Web 开发语言

用于开发 CGI 的几种常用语言有 C、PERL、SH、TCL 等, 除了 C 是编译型语言外, 其他几种都是解释型的开发语言。

以下只是举一些简单的例子加以说明, 至于每种语言的细节, 并不在本书的讨论范围内。

1. SH

```
#!/bin/sh
echo Content-type: text/plain
echo
echo CGI/1.0 test script report:
echo
echo argc is $#. argv is "$*".
echo
echo SERVER_SOFTWARE = $SERVER_SOFTWARE
echo SERVER_NAME = $SERVER_NAME
echo GATEWAY_INTERFACE = $GATEWAY_INTERFACE
echo SERVER_PROTOCOL = $SERVER_PROTOCOL
echo SERVER_PORT = $SERVER_PORT
echo REQUEST_METHOD = $REQUEST_METHOD
echo HTTP_ACCEPT = "$HTTP_ACCEPT"
echo PATH_INFO = $PATH_INFO
echo PATH_TRANSLATED = $PATH_TRANSLATED
echo SCRIPT_NAME = $SCRIPT_NAME
echo QUERY_STRING = $QUERY_STRING
echo REMOTE_HOST = $REMOTE_HOST
echo REMOTE_ADDR = $REMOTE_ADDR
echo REMOTE_USER = $REMOTE_USER
echo CONTENT_TYPE = $CONTENT_TYPE
echo CONTENT_LENGTH = $CONTENT_LENGTH
```

这是用 UNIX 上的 SH 编写的简单的 CGI 程序, 用来检测 Web 服务器的 CGI 的环境变

量值，大家可以从这些环境变量入手，编写更深入的程序。

2. PERL (Practical Extraction and Report Language)

PERL 是一种优秀的解释型文本处理语言，现在已被一些 CGI 专家广泛使用。目前不仅有 UNIX 版本，还有 Windows 和其他操作系统的版本。其语法具有相同的规则。

3. C 语言

用 C 语言编写 CGI 程序类似于使用其他语言，只是需要编译成可执行文件。因此，用 C 语言进行 CGI 程序的编写具有更高的安全性。下面是一个用 C 语言编写的 CGI 小程序，仅供参考。

```
#include
#include
#define MAX_ENTRIES 10000
typedef struct {
    char *name;
    char *val;
} entry;
char *makeword(char *line, char stop);
char *fmakeword(FILE *f, char stop, int *len);
char x2c(char *what);
void unescape_url(char *url);
void plustospace(char *str);
main(int argc, char *argv[]) {
    entry entries[MAX_ENTRIES];
    register int x,m=0;
    int cl;
    if(strcmp(getenv("REQUEST_METHOD"),"POST")) exit(1);
    if(strcmp(getenv("CONTENT_TYPE"),"application/x-www-form-urlencoded")) exit(
        cl = atoi(getenv("CONTENT_LENGTH"));
        for(x=0;cl>0 && (!feof(stdin));x++) {
            m=x;
            entries[x].val = fmakeword(stdin,'&',&cl);
            plustospace(entries[x].val);
            unescape_url(entries[x].val);
            entries[x].name = makeword(entries[x].val,'=');
        }
        for(x=0; x <= m; x++)
            printf("%s %s\n",entries[x].name, entries[x].val);
    }
```

4. TCL

TCL 即 Tool Command Language，它是高级的跨平台 Script 语言，它与 PERL、C Shell 具有相似的原理和不同的语法。

8.5 JavaScript 脚本

JavaScript 语言的前身叫 LiveScript。自从 SUN 公司推出著名的 Java 语言之后，Netscape 公司引进了 SUN 公司有关 Java 的程序概念，将自己原有的 LiveScript 进行了重新设计，并改名为 JavaScript。

JavaScript 是一种基于对象和事件驱动并具有安全性能的脚本语言，JavaScript 可使网页变得生动。使用它的目的是与 HTML 超文本标识语言、Java 脚本语言一起实现在一个网页中链接多个对象，与网络客户进行交互，从而可以开发客户端的应用程序。它是通过嵌入在标准的 HTML 语言中实现的。

1. JavaScript 的优点

(1) 简单性。JavaScript 是一种脚本编写语言，它采用小程序段的方式实现编程。像其他脚本语言一样，JavaScript 同样也是一种解释性语言，它提供了一个简易的开发过程。它的基本结构形式与 C、C++、VB、Delphi 十分类似。但它不像这些语言，需要先编译，而是在程序运行过程中被逐行地解释。它与 HTML 标识结合在一起，从而方便用户的使用操作。

(2) 动态性。JavaScript 是动态的，它可以直接对用户或客户的输入做出响应，无须经过 Web 服务程序。它对用户的反映响应是采用以事件驱动的方式进行的。所谓事件驱动，是指在主页中执行了某种操作所产生的动作（称为“事件”），如按下鼠标、移动窗口、选择菜单等都可以视为事件，当事件发生后，可能会引起相应的事件响应。

(3) 跨平台性。JavaScript 依赖于浏览器本身，与操作环境无关，只要是能运行浏览器的计算机并支持 JavaScript 的浏览器就可以正确执行。

(4) 节省 CGI 的交互时间。随着 WWW 的迅速发展，有许多 WWW 服务器提供的服务要与浏览者进行交流，如确认浏览者的身份、需服务的内容等，这些工作通常由 CGI/PERL 编写相应的接口程序与用户进行交互来完成。很显然，通过网络与用户的交互过程一方面增加了网络的通信量，另一方面影响了服务器的服务性能。服务器为一个用户运行一个 CGI 时，需要一个进程为它服务，它要占用服务器的资源（如 CPU 服务、内存耗费等），如果用户填表出现错误，交互服务占用的时间就会相应增加。被访问的热点主机与用户交互越多，对服务器的性能影响就越大。

JavaScript 是一种基于客户端浏览器的语言，用户在浏览中填表、验证的交互过程是通过浏览器对调入 HTML 文档中的 JavaScript 源代码进行解释执行来完成的，即使必须调用 CGI 的部分，浏览器也只是将用户输入验证后的信息提交给远程的服务器，大大减少了服务器的开销。

2. JavaScript 和 Java 的区别

JavaScript 语言和 Java 语言是相关的，但它们之间的联系并不像想象中的那样紧密，二者的区别体现在以下几个方面。

(1) 它们是两个公司开发的两个不同的产品。Java 是 SUN 公司推出的新一代面向对象的程序设计语言，特别适合于 Internet 应用程序开发；JavaScript 是 Netscape 公司的产品，

其目的是为了扩展 Netscape Navigator 的功能而开发的一种可以嵌入 Web 页面中的基于对象和事件驱动的解释性语言。

(2) JavaScript 是基于对象的, 而 Java 是面向对象的, 即 Java 是一种真正的面向对象的语言, 即使是开发简单的程序, 也必须设计对象。JavaScript 是一种脚本语言, 它可以用来制作与网络无关的、与用户交互的复杂软件。它是一种基于对象和事件驱动的编程语言, 因而它本身提供了非常丰富的内部对象供设计人员使用。

(3) 两种语言在其浏览器中所执行的方式不一样。Java 的源代码在传递到客户端执行之前必须经过编译, 因而客户端上必须具有相应平台上的仿真器或解释器, 它可以通过编译器或解释器实现独立于某个特定的平台编译代码; JavaScript 是一种解释性编程语言, 其源代码在发往客户端执行之前不需要经过编译, 而是将文本格式的字符代码发送给客户端, 由浏览器解释执行。

(4) 两种语言所采取的变量是不一样的。Java 采用强类型变量检查, 即所有变量在编译之前必须进行声明; 而 JavaScript 中的变量声明采用弱类型, 即变量在使用前不需要进行声明, 而是解释器在运行时检查其数据类型。

(5) 代码格式不一样。Java 是一种与 HTML 无关的格式, 必须通过类似于在 HTML 中引用外媒体那样进行装载, 其代码以字节代码的形式保存在独立的文档中; 而 JavaScript 的代码是一种文本字符格式, 可以直接嵌入 HTML 文档中, 并且可动态装载, 编写 HTML 文档就像编辑文本文件一样方便。

(6) 嵌入方式不一样。在 HTML 文档中, 两种编程语言的标识不同, JavaScript 使用 `<script>...</script>` 来标识, 而 Java 使用 `<applet> ... </applet>` 来标识。

(7) 静态绑定和动态绑定。Java 采用静态联编, 即 Java 的对象引用必须在编译时进行, 以使编译器能够实现强类型检查; 而 JavaScript 采用动态联编, 即 JavaScript 的对象引用在运行时进行检查, 如果不经编译就无法实现对象引用的检查。

在目前流行的浏览器中, Netscape 公司的 Navigator 2.0 以上版本的浏览器都具有处理 JavaScript 源代码的能力。JavaScript 在其中实现了它的 1.0 版本, 并在后来的 Navigator 3.0 中实现了它的 1.1 版本, 在后来推出的 Navigator 4.0 (Communicator) 中, JavaScript 实现了它的 1.2 版本。

微软公司从它的 Internet Explorer 3.0 版开始支持 JavaScript。Microsoft 把自己实现的 JavaScript 规范称为 JScript。这个规范与 Netscape Navigator 浏览器中的 JavaScript 规范在基本功能和语法上是一致的, 但是在个别的对象实现方面有一定的差别, 这里特别需要予以注意。

3. JavaScript 的数据类型

JavaScript 有 6 种数据类型, 主要类型有 number、string、object 及 boolean, 其他两种类型为 null 和 undefined。

(1) 字符串类型 string: 字符串是用单引号或双引号来说明的 (使用单引号来输入包含引号的字符串), 如 "The cow jumped over the moon."。

(2) 数值数据类型 number: JavaScript 支持整数和浮点数。整数可以为正数、零或者负数; 浮点数可以包含小数点, 也可以包含一个 “e” (大小写均可, 在科学记数法中表示 “10

的幂”），或者同时包含这两项。

(3) **boolean** 类型：可能的 **boolean** 值有 **true** 和 **false**。这是两个特殊值，不能用作 1 和 0。

(4) **undefined** 数据类型：一个为 **undefined** 的值是指变量被创建后在给该变量赋值以前所具有的值。

(5) **null** 数据类型：**null** 值就是没有任何值，什么也不表示。

(6) **object** 类型：除了上面提到的各种常用类型外，对象也是 **JavaScript** 中的重要组成部分，这部分将在后面的章节中详细介绍。

在 **JavaScript** 中，变量用来存放脚本中的值，因此在需要用这个值的地方就可以用变量来代表，一个变量可以是数字、文本或其他一些东西。

JavaScript 是一种对数据类型变量要求不太严格的语言，所以不必声明每一个变量的类型。变量声明尽管不是必需的，但在使用变量之前先进行声明是一种良好的习惯。可以使用 **var** 语句来进行变量声明，如：

```
var men = true; //men 中存储的值为 boolean 类型
```

变量命名：**JavaScript** 是一种区分大小写的语言，因此将一个变量命名为 **computer** 和将其命名为 **Computer** 是不一样的。

另外，变量名称的长度是任意的，但必须遵循以下规则：

- (1) 第一个字符必须是字母（大小写均可）、下划线（**_**）或一个美元符（**\$**）。
- (2) 后续的字符可以是字母、数字、下划线或美元符。
- (3) 变量名称不能是保留字。

8.5.1 JavaScript 的语句及语法

1. JavaScript 的语句

JavaScript 所提供的语句分为以下几大类。

(1) 变量声明、赋值语句：**var**。

语法如下：

```
var 变量名称 [=初始值]
```

例如：

```
var computer = 32 //定义 computer 是一个变量，且有初值，为 32
```

(2) 函数定义语句：**function**，**return**。

语法如下：

```
function 函数名称（函数所带的参数）  
{  
    函数执行部分  
}
```

`return` 表达式 //`return` 语句指明返回的值

例如:

```
function square ( x )
{
    return x*x
}
```

(3) 条件和分支语句: `if...else`, `switch`。

`if...else` 语句完成程序流程块中的分支功能: 如果其中的条件成立, 则程序执行紧接着条件的语句或语句块; 否则程序执行 `else` 中的语句或语句块。

语法如下:

```
if(条件)
{
    执行语句 1
}else{
    执行语句 2
}
```

例如:

```
if (result == true)
{
    response = "你答对了! "
}else{
    response = "你错了! "
}
```

分支语句 `switch` 可以根据一个变量的不同取值采取不同的处理方法。

语法如下:

```
switch (expression)
{
    case label1: 语句串 1;
    case label2: 语句串 2;
    case label3: 语句串 3;
    ...
    default: 语句串 3;
}
```

如果表达式取的值与程序中提供的任何一条语句都不匹配, 则执行 `default` 中的语句。

(4) 循环语句: `for`, `for...in`, `while`, `break`, `continue`。

`for` 语句的语法如下:

`for` (初始化部分;条件部分;更新部分)

```
{
    执行部分...
}
```

只要循环的条件成立，循环体就被反复执行。

for...in 语句与 **for** 语句有一点不同，它循环的范围是一个对象的所有属性或是一个数组的所有元素。

for...in 语句的语法如下：

```
for (变量 in 对象或数组)
{
    语句...
}
```

while 语句所控制的循环不断地测试条件，如果条件始终成立，则一直循环，直到条件不再成立。

语法如下：

```
while (条件)
{
    执行语句...
}
```

break 语句结束当前的各种循环，并执行循环的下一条语句。

continue 语句结束当前的循环，并立即开始下一个循环。

(5) 对象操作语句：**with**，**this**，**new**。

with 语句的语法如下：

```
with (对象名称){
    执行语句
}
```

其作用为：如果想使用某个对象的许多属性或方法，只要在 **with()** 语句的 **()** 中写出这个对象的名称，然后在下面的执行语句中直接写这个对象的属性名或方法名就可以了。

new 语句是一种对象构造器，可以用 **new** 语句来定义一个新的对象。

语法如下：

```
新对象名称=new 真正的对象名
```

例如，可以这样定义一个新的日期对象：**var curr=new Date()**，然后，变量 **curr** 就有了 **Date** 对象的属性。

this 运算符总是指向当前的对象。

(6) 注释语句：**//**，**/*...*/**。

//用于单行注释。

/*....*/用于多行注释。

2. JavaScript 的对象及其属性和方法

在 JavaScript 中是基于对象的编程，而不是完全的面向对象的编程。

那么什么是对象呢？如果你学过一些 VB 的编程，对这个名词一定不会陌生。通俗地说，对象是变量的集合体，对象提供对于数据的一致性的组织手段，描述了一类事物的共同属性。

在 JavaScript 中，可以使用以下几种对象：

- (1) 由浏览器根据 Web 页面的内容自动提供的对象；
- (2) JavaScript 的内置对象，如 Date、Math 等；
- (3) 服务器上的固有对象；
- (4) 用户自定义的对象。

JavaScript 中的对象是由属性和方法两个基本元素构成的。对象的属性是指对象的背景色、长度、名称等；对象的方法是指对属性进行的操作，即一个对象自己所属的函数，如对对象取整、使对象获得焦点、使对象获得一个随机数等一系列操作。

举例来说，将汽车看成一个对象，则汽车的颜色、大小、品牌等称为属性，而发动、刹车、拐弯等就叫方法。

可以采用这样的方式来访问对象的属性：对象名称.属性名称。例如：

```
mycomputer.year=1996,mycomputer.owner="me"
```

可以采用这样的方式将对象的方法同函数联系起来：对象.方法名字=函数名字或对象.属性.方法名。例如：

```
this.display=display,document.writeln("this is method")
```

3. JavaScript 的事件处理

事件是浏览器响应用户交互操作的一种机制，JavaScript 的事件处理机制可以改变浏览器响应用户操作的方式，从而可以开发出具有交互性并易于使用的网页。

浏览器为了响应某个事件而进行的处理过程称为事件处理。

事件定义了用户与页面交互时产生的各种操作，如单击超级链接或按钮时，就产生一个单击（click）操作事件。浏览器在程序运行的大部分时间都在等待交互事件的发生，并在事件发生时自动调用事件处理函数，完成事件处理过程。

事件不仅可以在用户交互过程中产生，浏览器自己的一些动作也可以产生事件。例如，当载入一个页面时，会发生 load 事件；而卸载一个页面时，会发生 unload 事件。

归纳起来，必须使用的事件有三大类：

- (1) 引起页面之间跳转的事件，主要是超链接事件；
- (2) 事件浏览器自己引起的事件；
- (3) 事件在表单内部同界面对象的交互。

8.5.2 JavaScript 编程举例

【例 8-1】 在网页上用 JavaScript 实现数字时钟。

把这段程序放在<body>与</body>之间。

```
<script language="JavaScript">
<!--

function Time(){
if (!document.layers&&!document.all)
return
var Timer=new Date()
var hours=Timer.getHours()
var minutes=Timer.getMinutes()
var seconds=Timer.getSeconds()
var noon="AM"
if (hours>12){
noon="PM"
hours=hours-12
}
if (hours==0)
hours=12
if (minutes<=9)
minutes="0"+minutes
if (seconds<=9)
seconds="0"+seconds
//change font size here to your desire
myclock="<font size='4' face='Arial' color=blue>"+hours+":"+minutes+":"+seconds+" "+noon+"</b></font>"
if (document.layers){
document.layers.position.document.write(myclock)
document.layers.position.document.close()
}
else if (document.all)
position.innerHTML=myclock
setTimeout("Time()",1000)
}
//-->
</script>
<span id="position" style="position:absolute;left:441px;top:190px; width: 128px; height: 30px">
</span>
```

【例 8-2】 辨别浏览器。

把这段程序放在<body>与</body>之间

```
<script language="JavaScript">
<!--
document.write("<CENTER>您的浏览器是: " +navigator.appName + " " + navigator.appVersion
+"<CENTER>")
```

```
// -->  
</script>
```

网站服务器是建立在 socket 通信基础上的，因此要先了解什么是 socket。

8.6 socket 通信

socket 的英文原义是“孔”或“插座”。在这里作为 4BDS UNIX 的进程通信机制，取后一种意义。socket 非常类似于电话插座，以一个国家级电话网为例，电话的通话双方相当于相互通信的两个进程，区号是它的网络地址；区内一个单位的交换机相当于一台主机，主机分配给每个用户的局内号码相当于 socket 号。任何用户在通话之前，首先要占有一部电话机，相当于申请一个 socket；同时要知道对方的号码，相当于对方有一个固定的 socket。然后向对方拨号呼叫，相当于发出连接请求（假如对方不在同一区内，还要拨对方的区号，相当于给出网络地址）。假如对方在场并空闲（相当于通信的另一主机开机且可以接受连接请求），拿起电话话筒，双方就可以正式通话了，相当于连接成功。双方通话的过程，是一方向电话机发出信号和对方从电话机接收信号的过程，相当于向 socket 发送数据和从 socket 接收数据。通话结束后，一方挂起电话机相当于关闭 socket，撤销连接。

在电话系统中，一般用户只能感受到本地电话机和对方电话号码的存在，建立通话的过程、语音传输的过程及整个电话系统的技术细节对他都是透明的，这也与 socket 机制非常相似。socket 利用网间网通信设施实现进程通信，但它对通信设施的细节毫不关心，只要通信设施能提供足够的通信能力，它就满足了。

至此，我们对 socket 进行了直观的描述。抽象出来，socket 实质上提供了进程通信的端点。进程通信之前，双方首先必须各自创建一个端点，否则是无法建立联系并相互通信的。正如打电话之前，双方必须各自拥有一台电话机一样。在网间网内部，每一个 socket 都用一个半相关描述：

（协议，本地地址，本地端口）

一个完整的 socket 有一个本地唯一的 socket 号，由操作系统分配。

最重要的是，socket 是面向客户端/服务器模型而设计的，针对客户端和服务程序提供不同的 socket 系统调用。客户端随机申请一个 socket（相当于一个想打电话的人可以在任何一台入网电话上拨号呼叫），系统为其分配一个 socket 号；服务器拥有全局公认的 socket，任何客户端都可以向它发出连接请求和信息请求（相当于一个被呼叫的电话拥有一个呼叫方知道的电话号码）。

socket 利用客户端/服务器模式巧妙地解决了进程之间建立通信连接的问题。服务器 socket 为全局所公认非常重要。读者不妨考虑一下，两个完全随机的用户进程之间如何建立通信？假如通信双方没有任何一方的 socket 固定，则好比打电话的双方彼此不知道对方的电话号码一样，要通话是不可能的。

socket 接口是访问 Internet 使用最广泛的方法。如果有一台刚配好 TCP/IP 协议的主机，其 IP 地址是 202.120.127.201，此时在另一台主机或同一台主机上执行“ftp 202.120.

127.201”，显然无法建立连接，因为“202.120.127.201”这台主机没有运行 FTP 服务软件。同样，在另一台或同一台主机上运行浏览软件，如 Netscape，输入“http://202.120.127.201”，也无法建立连接。现在，如果在这台主机上运行一个 FTP 服务软件（该软件将打开一个 socket，并将其绑定到 21 端口），再在这台主机上运行一个 Web 服务软件（该软件将打开另一个 socket，并将其绑定到 80 端口）。这样，在另一台主机或同一台主机上执行“ftp 202.120.127.201”，FTP 客户软件将通过 21 端口来呼叫主机上由 FTP 服务软件提供的 socket，与其建立连接并对话。而在 Netscape 中输入“http://202.120.127.201”时，将通过 80 端口来呼叫主机上由 Web 服务软件提供的 socket，与其建立连接并对话。

Internet 上有很多这样的主机，这些主机一般运行了多个服务软件，同时提供几种服务。每种服务都打开一个 socket，并绑定到一个端口上，不同的端口对应不同的服务。socket 正如其英文原义那样，像一个多孔插座。一台主机犹如布满各种插座的房间，每个插座都有一个编号，有的插座提供 220V 交流电，有的提供 110V 交流电，有的则提供有线电视节目。客户端软件将插头插到不同编号的插座，就可以得到不同的服务。

一个人要能收到别人打给他的电话，首先要装上一部电话。同样，在进行通信前必须先建立 socket 以侦听线路。这个过程包含几个步骤，如图 8-5 所示。

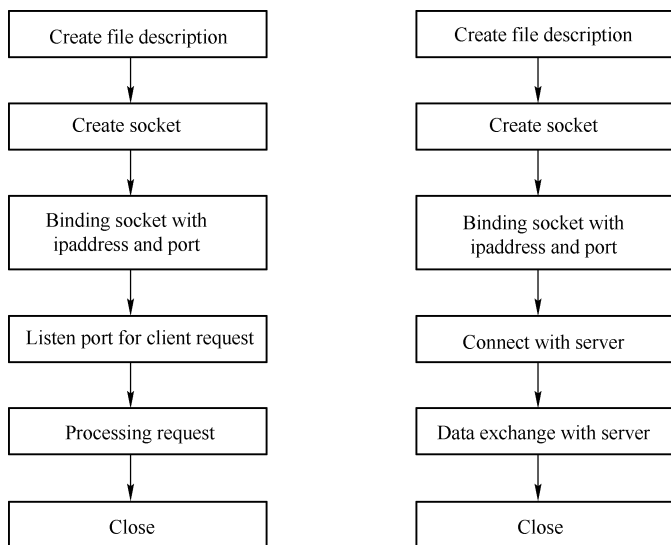


图 8-5 socket 通信过程

(1) 建立一个新的 socket，就像先装上电话一样。socket()命令可以完成这个工作。因为 socket 有几种类型，所以需注明要建立的类型。还要做一个选择，即 socket 的地址格式。如同电话有音频和脉冲两种形式一样，socket 有两个最重要的选项，为 AF_UNIX 和 IAF_INET。AF_UNIX 像 UNIX 路径名一样可以识别 socket，这种形式对于在同一台机器上的 IPC 很有用；而 AF_INET 是像 192.9.200.10 这样被点号隔开的 4 个十进制数字的地址格式。除了机器地址以外，还可以利用端口号来允许每台机器上有多个 AF_INET socket。此处偏重于 AF_INET 方式，因为其很有用并被广泛使用。

另外一个必须提供的参数是 `socket` 的类型。常用的 `socket` 类型有两种：流式 `socket` (`SOCK_STREAM`) 和数据报式 `socket` (`SOCK_DGRAM`)。流式 `socket` 是一种面向连接的 `socket`，针对于面向连接的 TCP 服务应用；数据报式 `socket` 是一种无连接的 `socket`，对应于无连接的 UDP 服务应用。下面将介绍流式 `socket`，因为其常见并易于使用。

(2) 在建立 `socket` 后，要提供 `socket` 侦听的地址。就像还需要电话号码来打电话一样。`bind()` 函数负责完此任务。

(3) `SOCK_STREAM socket` 让连接请求形成一个队列。如果用户忙于处理一个连接，别的连接请求将一直等待，直到该连接处理完毕。`listen()` 函数用来设置最大不被拒绝的请求数。

`socket` 函数的原型为 `int socket(int domain, int type, int protocol)`，`domain` 参数指明所使用的协议族，通常为 `PF_INET`，表示互联网协议族 (TCP/IP 协议族)；`type` 参数指定 `socket` 的类型——`SOCK_STREAM` 或 `SOCK_DGRAM`。

`socket()` 调用返回一个整型 `socket` 描述符，可以在后面的调用中使用它。`socket` 描述符是一个指向内部数据结构的指针，它指向描述符的表入口。调用 `socket` 函数时，`socket` 执行体将建立一个 `socket`，实际上“建立一个 `socket`”意味着为一个 `socket` 数据结构分配存储空间。`socket` 执行体管理描述符表。两个网络程序之间的一个网络连接包括 5 种信息：通信协议、本地协议地址、本地主机端口、远端主机地址和远端协议端口。

下面的代码说明如何利用 `socket()`、`bind()` 和 `listen()` 函数建立连接并接收数据。

8.6.1 TCP Socket 编程举例

服务器程序：`tcpsocketserver.c`。

```
#include <stdio.h>
#include <strings.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PORT 1235
#define BACKLOG 1
main()
{
    int listenfd, connectfd;
    struct sockaddr_in server;
    struct sockaddr_in client;
    int sin_size;

    if((listenfd = socket(AF_INET, SOCK_STREAM, 0)) == -1){
        perror("Creating socket failed.");
        exit(1);
    }
```

```

}
int opt=SO_REUSEADDR;
setsockopt(listenfd,SOL_SOCKET,SO_REUSEADDR,&opt,sizeof(opt));
bzero(&server,sizeof(server));

/*-----
struct sockaddr_in {
short int sin_family; /* 地址族 */
unsigned short int sin_port; /* 端口号 */
struct in_addr sin_addr; /* IP 地址 */
unsigned char sin_zero[8]; /* 填充 0, 以保持与 struct sockaddr 大小相等 */
};
sin_zero 用来将 sockaddr_in 结构填充到与 struct sockaddr 相等的长度, 可以用 bzero()函数将其设置为零。
-----*/

```

```

server.sin_family=AF_INET;
server.sin_port=htons(PORT);
server.sin_addr.s_addr=htonl(INADDR_ANY);

if(bind(listenfd,(struct sockaddr*)&server,sizeof(struct sockaddr)) == -1){
perror("Bind error");
exit(1);
}

```

```

/*-----
使用 bind 函数时, 可以用下面的赋值实现自动获得本机 IP 地址和获取端口号:
server.sin_port = htons(PORT); /* 设置端口号 */
server.sin_addr.s_addr = INADDR_ANY; /* 填入本机 IP 地址 */
通过将 server.sin_addr.s_addr 设置为 INADDR_ANY, 系统自动填入本机 IP 地址。
注意, 在使用 bind 函数时需要将 sin_port 和 sin_addr 转换成网络字节优先顺序。

```

计算机数据存储有两种字节优先顺序: 高位字节优先和低位字节优先。Internet 中数据以高位字节优先顺序在网络上传输, 所以对于在内部是以低位字节优先方式存储数据的机器, 在 Internet 上传输数据时需要进行转换, 否则就会出现数据不一致的情况。

下面是几个字节顺序转换函数。

```

htonl(): 把 32 位值从主机字节序转换成网络字节序。
htons(): 把 16 位值从主机字节序转换成网络字节序。
ntohl(): 把 32 位值从网络字节序转换成主机字节序。
ntohs(): 把 16 位值从网络字节序转换成主机字节序。

```

```

-----*/

if(listen(listenfd,BACKLOG) == -1){
perror("listen() error\n");

```

```
exit(1);
}
```

```
/*-----*/
```

Listen 函数使 socket 处于被动的监听模式，并为该 socket 建立一个输入数据队列，将到达的服务请求保存在此队列中，直到程序处理它们。

sockfd 是 socket 系统调用返回的 socket 描述符；backlog 指定在请求队列中允许的最大请求数，进入的连接请求将在队列中等待 accept()（参考下文）。backlog 对队列中等待服务的请求数目进行了限制，大多数系统默认值为 20。如果一个服务请求到来时，输入队列已满，则该 socket 将拒绝连接请求，客户端将收到一个出错信息。当出现错误时 listen 函数返回-1，并置相应的 error 错误码。

```
-----*/
```

```
sin_size=sizeof(struct sockaddr_in);
```

```
if((connectfd = accept(listenfd,(struct sockaddr *)&client,&sin_size)) == -1){
perror("accept() error\n");
exit(1);
}
```

```
/*-----*/
```

accept()函数让服务器接收客户端的连接请求。在建立好输入队列后，服务器就调用 accept()函数，然后睡眠并等待客户端的连接请求。

```
int accept(int sockfd, void *addr, int *addrlen);
```

sockfd 是被监听的 socket 描述符；addr 通常是一个指向 sockaddr_in 变量的指针，该变量用来存放提出连接请求服务的主机的信息（某台主机从某个端口发出该请求）；addrlen 通常为一个指向值为 sizeof(struct sockaddr_in)的整型指针变量。出现错误时 accept()函数返回-1 并置相应的 error 值。当 accept()函数监视的 socket 收到连接请求时，socket 执行体将建立一个新的 socket，执行体将这个新 socket 和请求连接进程的地址联系起来，收到服务请求的初始 socket 仍可以继续以前的 socket 上监听，同时可以在新的 socket 描述符上进行数据传输操作。

```
-----*/
```

```
printf("your got a connection from %s\n",inet_ntoa(client.sin_addr));
send(connectfd,"welcome to my server.\n",22,0);
```

```
/*-----*/
```

send()函数的原型为：

```
int send(int sockfd, const void *msg, int len, int flags);
```

sockfd 是用来传输数据的 socket 描述符；msg 是一个指向要发送数据的指针；len 是以字节为单位的数据长度；flags 一般情况下设置为 0。

send()函数返回实际上发送出的字节数，可能会少于希望发送的数据。在程序中应该将 send()的返回值与欲发送的字节数进行比较。当 send()返回值与 len 不匹配时，应该对这种情况进行处理。

```
-----*/
```

```

close(connectfd);
close(listenfd);
}
/*-----
当所有的数据操作都结束以后，可以调用 close()函数来释放该 socket，从而停止在该 socket 上
的任何数据操作：
close(sockfd);
-----*/

```

客户端程序：tcpsocketclient.c。

```

#include <stdio.h>
#include <strings.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>

#define PORT 1235
#define MAXDATASIZE 100

int main(int argc, char *argv[])
{
    int fd, numbytes;
    char buf[MAXDATASIZE];
    struct hostent *he;
    struct sockaddr_in server;

    if(argc!=2){
        printf("Usage: %s<IP Address>\n", argv[0]);
        exit(1);
    }

    /*if((he=gethostbyname(argv[1]))==NULL){
        printf("gethostbyname() error\n");
        exit(1);
    } */

    if((fd=socket(AF_INET, SOCK_STREAM, 0))==1){
        printf("socket() error\n");
    }

    /*-----
建立 socket。

```

```

-----*/

bzero(&server,sizeof(server));
server.sin_family=AF_INET;
server.sin_port=htons(PORT);
//server.sin_addr=(he->h_addr);
/*if (inet_aton(argv[1], (struct in_addr *) &server.sin_addr.s_addr) == 0) {
perror(argv[1]);
exit(1);
}*/
server.sin_addr.s_addr = inet_addr(argv[1]);

/*-----
将 IP 地址赋给 server 的 socket 结构的成员。
-----*/

printf("address created\n");

if(connect(fd,(struct sockaddr *)&server,sizeof(struct sockaddr)) == -1){
printf("connect() error\n");
exit(1);
}

/*-----
和远程服务器建立连接。
-----*/

if ((numbytes=recv(fd,buf,MAXDATASIZE,0))==-1){
printf("recv() error\n");
exit(1);
}

/*-----
recv()函数的原型为：
int recv(int sockfd,void *buf,int len,unsigned int flags);
sockfd 是接收数据的 socket 描述符；buf 是存放接收数据的缓冲区；len 是缓冲的长度；flags 也
被设置为 0。recv()返回实际上接收的字节数，当出现错误时，返回-1 并置相应的 err 值。
-----*/

buf[numbytes]='\0';
printf("Server Message:%s\n",buf);
/*-----
输出接收的信息。
-----*/

```

```

-----*/
close(fd);
}

```

下面做个实验：

将 tcpsocketserver.c 和 tcpsocketclient.c 分别编译为可执行文件。

```

gcc -o tcpsocketclient tcpsocketclient.c
gcc -o tcpsocketserver tcpsocketserver.c
arm-linux-gcc -o tcpsocketserver_arm tcpsocketserver.c

```

首先在 x86 下测试，然后在 ARM 上测试。

在 x86 下测试，假设其 IP 地址是 192.168.1.80。

开启一个终端输入：

```
./tcpsocketserver
```

开启另外一个终端输入：

```
./tcpsocketclient 192.168.12.80
```

结果如下：

```

[root@localhost socket]# ./tcpsocketclient 192.168.12.80
address created
Server Message:welcome to my server.

```

假设服务器（x86）的 IP 地址为 192.168.12.80，开发板（ARM）的 IP 地址为 192.168.12.8。

首先在开发板上运行服务程序，结果如下：

```

[root@51Board mnt]# ./tcpsocketserver_arm
your got a connection from 192.168.12.80
[root@51Board mnt]#

```

然后在 x86 上运行程序，结果如下：

```

-----
Usage: ./tcpsocketclient<IP Address>
[root@localhost socket]# ./tcpsocketclient 192.168.12.8
address created
Server Message:welcome to my server.

```

8.6.2 UDP Socket 编程举例

UDP/IP 提供了无连接的传输层协议：UDP（User Datagram Protocol，用户数据报协议）。UDP 与 TCP 有很大的区别，因为无连接的 socket 编程与面向连接的 socket 编程也有很大的差异。由于不用建立连接，因此每个发送和接收的数据报都包含了发送方和接收方的地址信息。在发送和接收数据之前，要先建立一个数据报方式的套接字。该 socket 的类型

为 SOCK_DGRAM, 用如下的调用产生:

```
sockfd=socket(AF_INET, SOCK_DGRAM, 0);
```

由于不需要建立连接, 因此产生 socket 后就可以直接发送和接收了。当然, 要接收数据报也必须绑定一个端口, 否则发送方无法得知要发送到哪个端口。sendto 和 recvfrom 两个系统调用分别用于发送和接收数据报, 其调用格式为:

```
int sendto(int s, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen);
int recvfrom(int s, void *buf, int len, unsigned int flags, struct sockaddr *from, int fromlen);
```

其中, s 为所使用的 socket; msg 和 buf 分别为发送和接收的缓冲区指针; len 为缓冲区的长度; flags 为选项标志, 此处还用不到, 设置为 0 即可; to 和 from 是发送的目的地址和接收的来源地址, 包含了 IP 地址和端口信息; tolen 和 fromlen 分别是 to 和 from 这两个 socket 地址结构的长度。上述两个函数的返回值就是实际发送和接收的字节数, 返回-1 表示出错。

图 8-6 描述的是通信双方都绑定自己地址端口的情形, 但在某些情况下, 也可能有一方不用绑定地址和端口。

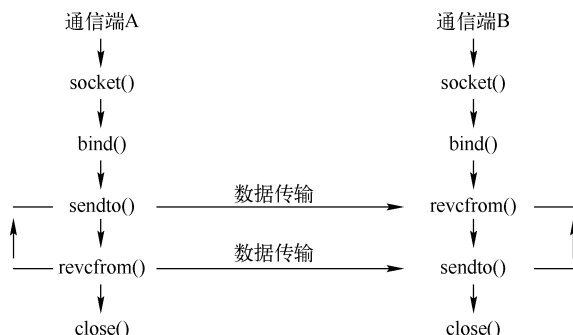


图 8-6 UDP 通信

不绑定的一方其地址和端口由内核分配。由于对方无法预先知道不绑定的一方的端口和 IP 地址（假设主机有多个端口, 这些端口分配了不同的 IP 地址）, 因此只能由不绑定的一方先发出数据报, 对方根据收到的数据报中的来源地址就可以确定回送数据报所需要的发送地址了。显然, 在这种情况下对方必须绑定地址和端口, 并且通信只能由非绑定方发起。与 read()和 write()相似, 进程阻塞在 recvfrom()和 sendto()中也会发生。但是与 TCP 方式不同的是, 接收到一个字节数为 0 的数据报是有可能的, 应用程序完全可以将 sendto()中的 msg 设为 NULL, 同时将 len 设为零。

下面是一个基于以上原理分析的 UDP 编程的例子:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdio.h>
```

```

#define BUFLen 255
int main(int argc, char **argv)
{
    struct sockaddr_in peeraddr, /*存放谈话对方 IP 和端口的 socket 地址*/
    localaddr; /*本端 socket 地址*/
    int sockfd;
    char recmsg[BUFLen+1];
    int socklen, n;
    if(argc!=5){
        printf("%s <dest IP address> <dest port> <source IP address> <source port>\n", argv[0]);
        exit(0);
    }
    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if(sockfd<0){
        printf("socket creating err in udptalk\n");
        exit(1);
    }
    /*-----*/

```

socket 函数的原型为 `int socket(int domain, int type, int protocol)`, domain 指明所使用的协议族, 通常为 `PF_INET`, 表示互联网协议族 (TCP/IP 协议族); type 参数指定 socket 的类型, 这里是 `SOCK_DGRAM`。

socket()调用返回一个整型 socket 描述符, 可以在后面的调用中使用它。

```

-----*/

```

```

socklen = sizeof(struct sockaddr_in);
memset(&peeraddr, 0, socklen);
peeraddr.sin_family=AF_INET;
peeraddr.sin_port=htons(atoi(argv[2]));
/*-----*/

```

将对方的端口赋值给 peeraddr。

```

-----*/

```

```

if(inet_pton(AF_INET, argv[1], &peeraddr.sin_addr)<=0){
    printf("Wrong dest IP address!\n");
    exit(0);
}
/*-----*/

```

将对方的 IP 地址赋值给 peeraddr。

```

-----*/

```

```

memset(&localaddr, 0, socklen);
localaddr.sin_family=AF_INET;
if(inet_pton(AF_INET, argv[3], &localaddr.sin_addr)<=0){
    printf("Wrong source IP address!\n");
    exit(0);
}
/*-----*/

```



```
将本地的 IP 地址赋值给 peeraddr。
-----*/

}
localaddr.sin_port=htons(atoi(argv[4]));
/*-----
将本地的端口赋值给 localaddr。
-----*/

if(bind(sockfd, &localaddr, socklen)<0){
printf("bind local address err in udptalk!\n");
exit(2);
}
/*-----
将本地的 IP 地址和端口与 socket 的文件描述符建立绑定。
-----*/

if(fgets(recmsg, BUFLen, stdin) == NULL) exit(0);
/*-----
从终端读取输入的字符。
-----*/

if(sendto(sockfd, recmsg, strlen(recmsg), 0, &peeraddr, socklen)<0){
printf("sendto err in udptalk!\n");
exit(3);
}
/*-----
将读取的输入字符发送给对方。
-----*/

for(;;){
/*recv&send message loop*/
n = recvfrom(sockfd, recmsg, BUFLen, 0, &peeraddr, &socklen);
if(n<0){
printf("recvfrom err in udptalk!\n");
exit(4);
}else{
/*成功接收到数据报*/
recmsg[n]=0;
printf("peer:%s", recmsg);
}
/*-----
接收对方发来的信息并打印。
-----*/

if(fgets(recmsg, BUFLen, stdin) == NULL) exit(0);
```

```

if(sendto(sockfd, recmsg, strlen(recmsg), 0, &peeraddr, socklen)<0){
    printf("sendto err in udptalk!\n");
    exit(3);
}
}
}
}

```

以上程序的文件名是udpsocket.c,编译为ARM和x86版本的执行文件为:

```

gcc -o udpsocket udpsocket.c
arm-linux-gcc -o udpsocket_arm udpsocket.c

```

假设主机的 IP 地址为 192.168.12.80, PXA270 的 IP 地址为 192.168.12.8。
在主机中输入信息后回车,接着在开发板终端输入信息并回车,将出现如下界面。
开发板信息:

```

[root@51Board mnt]# ./udpsocket_arm 192.168.12.80 1234 192.168.12.8 1235
i am super wang
peer:who are you
thank you
peer:oh, i see

```

主机信息:

```

[root@localhost socket]# ./udpsocket 192.168.12.8 1235 192.168.12.80 1234
who are you
peer:i am super wang
oh, i see
peer:thank you

```

在了解了 socket 的基础上,接下来研究最简单的 Web 服务器,先看看具体的 Client 和 Server 的交互过程

8.6.3 HTTP 请求中 Client 与 Server 的交互过程

用户在 Client (浏览器) 中输入一个 URL, 浏览器将用户输入的 URL 与 Cookie 等一起生成一个 HTTP request 发送给 Server (Client 通过 DNS 解析找到 Server)。HTTP request 包括 HTTP header 和 HTTP body 两个部分, 其中 HTTP header 中有 URL 和 Cookie 等。而 HTTP body 则分为三种情况。

(1) 如果 HTTP request 为 GET 模式, 则 HTTP body 为空。

(2) 如果 HTTP request 为 POST 模式, 则 HTTP body 为 URL 参数 (也就是 URL 中?之后的那部分)。

(3) 如果用户需要上传文件 (此时 HTTP request 一般为 POST 模式), 则 HTTP body 为 multipart 格式。

1. 从 Client 到 Server

浏览器一般是图形界面的, 因此我们并不了解在这华丽表面后所发生的一切。当单击一个链接时, 浏览器首先找到站点的 IP 地址, 这是通过 DNS 来实现的。在找到 IP 地址后就可以建立 TCP 连接了, 连接建立后就可以发送请求了, 但是这个请求是什么样子的呢?

现在假设从 `www.armv.cn/index2.html` 单击了 `www.cleanrobot.com/paper/`，这时浏览器会发出下面的请求：

```
Get /mattmarg/ HTTP/1.0
User-Agent: Mozilla/2.0 (Macintosh; I; PPC)
Accept: text/html; */*
//本文转自 C++Builder 研究 - http://www.ccrun.com/article.asp?i=812&d=6p04g2
Cookie: name = value
Referer: http://www.webmonkey.com/html/96/47/index2a.html
Host: www.grippy.org
```

第一行称为请求，它告诉服务器从 `mattmarg` 取得文件，这里的目录一般是要加“/”的。下面几行通知服务器所使用的浏览器是什么类型，所接收的数据是什么类型。如果以前访问过这个站点，则站点可能发送了 `Cookie`，如果已经有了一个这样的 `Cookie`，则浏览器会将这个 `Cookie` 返回给服务器。`Referer` 行通知服务器用户是从哪一页到达此页的。

2. 从 Server 到 Client

Server 通过在服务器监听 80 端口收到 Client 发过来的 HTTP request，取出 URL 并将其映射到本地的文件，如果是 HTML 文件则直接生成一个 HTTP request 返回给 Client，如果是 PHP 文件则由 PPH 解释生成 HTTP request 返回给 Client。此时 HTTP request 的 HTTP deader 部分主要包含 Location（跳转指令）及 Cookie 等，HTTP body 包含 HTML 内容（当然也可以是任意其他格式的数据，如图片等）。

接下来服务器就要返回文件了，每次服务器返回文件时都要返回一个 HTTP/1.0 响应，同时带有状态码，在此之后是一些描述内部的头信息。下面就是一个响应：

```
HTTP/1.0 200 Found
Date: Mon, 10 Feb 1997 23:48:22 GMT
Server: Apache/1.1.1 HotWired/1.0
Content-type: text/html
Last-Modified: Tues, 11 Feb 1997 22:45:55 GMT
```

不同的数据可能返回不同的 Content-type，而不同的内容需要不同的 Content-type，因此有时这个过程是很慢的。

8.6.4 一个简单的 Web 服务器例子

图 8-7 是 Web 和客户端之间的 HTTP 交互，演示了针对客户端的请求如何将目录、文件等输出到客户端，这对理解嵌入式 Web 内部的工作很重要。

Web 上的主程序建立 TCP 类型的 socket，在 80 端口监听连接请求。接收到连接请求后将请求传送给连接处理模块进行处理，并继续进行监听。

1. Web 的功能分配

1) 发送当前目录文件列表信息

将服务器当前目录下所有文件的信息发送给客户端，信息包括文件名、大小、日期。

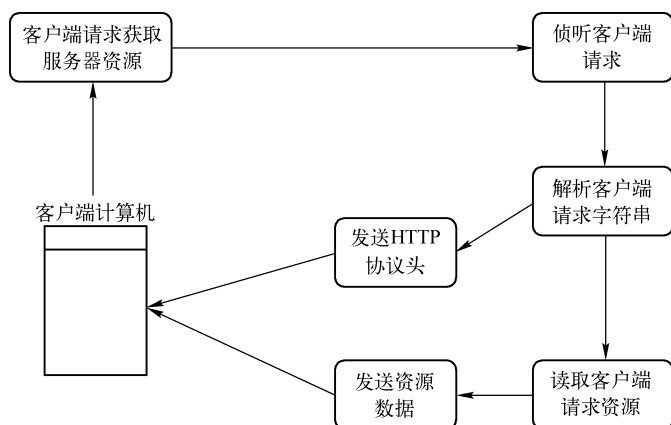


图 8-7 Web 和客户端之间的 HTTP 交互

2) 发送 HTML 类型文件

将客户端请求的 HTML 类型文件发送给客户端。

3) 发送纯文本文件

将客户端请求的纯文本文件发送给客户。

4) 发送 JPG 图像文件

将客户端请求的 JPG 图像文件发送给客户端。

5) 发送 GIF 图像文件

将客户端请求的 GIF 图像文件发送给客户端。

6) 解析客户端请求 HTTP 协议头

分析客户端的请求，包括：

- (1) 空格处理；
- (2) 解析客户端请求命令；
- (3) 解析客户端请求的资源名；
- (4) 解析客户端请求的资源类型。

7) 客户端连接处理

解析 `referrer` 和 `content_length` 字段值并调用客户端请求解析函数。

8) 发送 HTTP 协议数据头

根据发送文件类型发送相应的 HTTP 协议头信息。

9) 解析客户端请求

函数名：int ParseReq(FILE *f, char *r)

参数：参数 1——文件流 FILE 结构指针，用于表示客户端连接的文件流指针；

参数 2——字符串指针，待解析的字符串。

10) 发送 HTTP 协议数据头

函数名：int PrintHeader(FILE *f, int content_type)

参数：参数 1——文件流 FILE 结构指针，用于表示客户端连接的文件流指针，用于写入 HTTP 协议数据头信息；

参数 2——信息类型，用于确定发送的 HTTP 协议数据头信息。

11) 发送当前目录文件列表信息

函数名: int DoDir(FILE *f, char *name)

参数: 参数 1——文件流 FILE 结构指针，用于表示客户端连接的文件流指针，用于写入目录文件信息数据；

参数 2——目录名，表示客户端请求的目录信息。

12) 发送 HTML 文件内容

函数名: int DoHTML(FILE *f, char *name)

参数: 参数 1——文件流 FILE 结构指针，用于表示客户端连接的文件流指针，用于写入文件信息数据；

参数 2——客户端请求的文件名。

13) 发送纯文本 (TXT) 文件内容

函数名: int DoText(FILE *f, char *name)

参数: 参数 1——文件流 FILE 结构指针，用于表示客户端连接的文件流指针，用于写入文件信息数据；

参数 2——客户端请求的文件名。

14) 发送 JPEG 图像文件内容

函数名: int DoJpeg(FILE *f, char *name)

参数: 参数 1——文件流 FILE 结构指针，用于表示客户端连接的文件流指针，用于写入文件信息数据；

参数 2——客户端请求的文件名。

15) 发送 GIF 图像文件内容

函数名: int DoGif(FILE *f, char *name)

参数: 参数 1——文件流 FILE 结构指针，用于表示客户端连接的文件流指针，用于写入文件信息数据；

参数 2——客户端请求的文件名。

2. Web 服务器内部模块设计

1) 主程序设计

(1) 功能说明: 系统的总入口，也是系统的主要控制函数。分别完成如下功能:

- 建立环境设置；
- 设置信号处理方式；
- 建立侦听 TCP 流方式并绑定 80 端口；
- 建立连接侦听及客户 socket 连接处理调用主循环。

(2) 算法流程图: 算法流程图如图 8-8 所示。

(3) 命令行输入处理: 用户在命令行输入参数 -i，则将客户端输入文件描述字设为 0，即标准输入，用于在本机进行测试。其他输入全部忽略。

2) 客户端连接处理模块设计

(1) 功能说明: 用于初步处理客户端的连接请求，并将请求信息传递给客户端请求解

析函数处理。

(2) 算法：具体算法如图 8-9 所示。

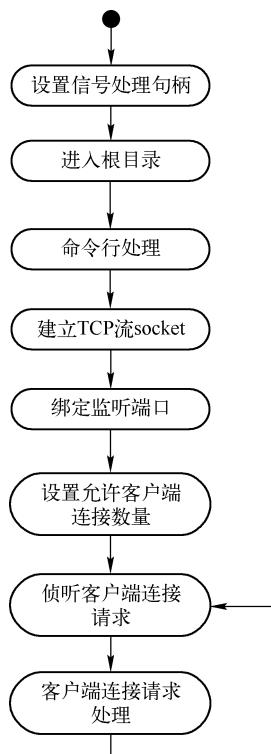


图 8-8 算法流程图

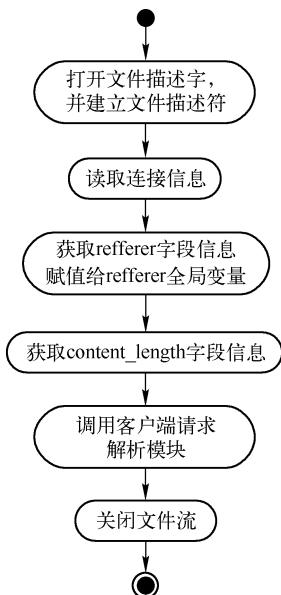


图 8-9 客户端连接处理算法

3) 发送 HTTP 协议数据头模块设计

(1) 功能说明：根据参数的不同，发送不同的 HTTP 协议头信息。

(2) 算法：函数定义为 `int PrintHeader(FILE *f, int content_type)`。

① 发送请求成功信息：HTTP/1.0 200 OK。

② 根据文档类型发送相应的信息：`fprintf()`。函数中的第一个参数 `f` 为客户端连接文件流句柄。

```

switch (content_type)
{
case 't':
fprintf(f,"Content-type: text/plain\n");
break;
case 'g':
fprintf(f,"Content-type: image/gif\n");
BREAK;
case 'j':
fprintf(f,"Content-type: image/jpeg\n");
break;
}
  
```

```
case 'h':
    fprintf(f,"Content-type: text/html\n");
    break;
}
```

(3) 发送服务器信息:

```
fprintf(f,"Server: AMRLinux-httpd 0.2.4\n");
```

4) 发送文件过期为永不过期

```
fprintf(f,"Expires: 0\n");
```

图 8-10 是 HTTP 协议数据头模块。

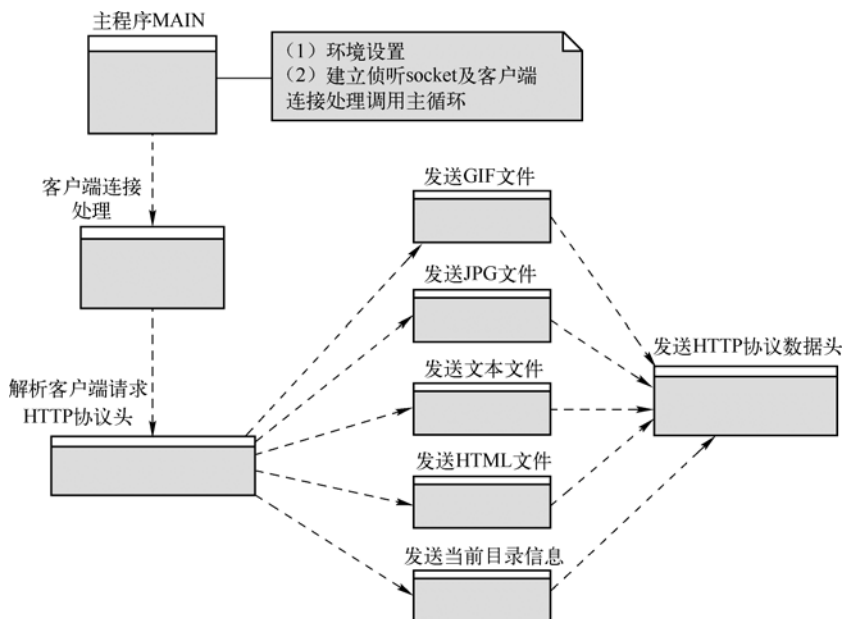


图 8-10 HTTP 协议数据头模块

将 httpd.c 编译为 httpd，下载到开发板运行，在开发板的 httpd 当前目录下存放 index.html，用 PC 的 IE 访问开发板：http://192.168.xx.xx/index.html。

这个例子很简单，服务器无法运行 CGI 脚本，在嵌入式系统中不实用，为此，需要可以支持运行 CGI 的服务器。

8.7 嵌入式 Web 服务器 Boa 的特点

Boa 是一款单任务的 HTTP 服务器，与其他传统的 Web 服务器不同的是，当有连接请求到来时，它并不为每个连接单独创建进程，也不通过复制自身进程来处理多连接，而是通过建立 HTTP 请求列表来处理多路 HTTP 连接请求，同时它只为 CGI 程序创建新的进程，这

样就在最大程度上节省了系统资源，这对嵌入式系统来说至关重要。同时，它还具有自动生成目录、自动解压文件等功能，因此 Boa 有很高的 HTTP 请求处理速度和效率，在嵌入式系统中具有很高的应用价值。

8.7.1 Boa 的功能实现

嵌入式 Web 服务器 Boa 和普通 Web 服务器一样，能够完成接收客户端请求、分析请求、响应请求、向客户端返回请求结果等任务。它的工作过程主要包括：

- (1) 完成 Web 服务器的初始化工作，如创建环境变量、创建 TCP 套接字、绑定端口、开始侦听、进入循环结构，以及等待接收客户端浏览器的连接请求；
- (2) 当有客户端连接请求时，Web 服务器负责接收客户端请求，并保存相关请求信息；
- (3) 在接收到客户端的连接请求之后，分析客户端请求，解析出请求的方法、URL 目标、可选的查询信息及表单信息，同时根据请求做出相应的处理；
- (4) Web 服务器完成相应处理后，向客户端浏览器发送响应信息，关闭与客户端的 TCP 连接。

嵌入式 Web 服务器 Boa 根据请求方法的不同做出不同的响应。如果请求方法为 HEAD，则直接向浏览器返回响应首部；如果请求方法为 GET，则在返回响应首部的同时，将客户端请求的 URL 目标文件从服务器上读出，并且发送给客户端浏览器；如果请求方法为 POST，则将客户端发送过来的表单信息传送给相应的 CGI 程序，作为 CGI 的参数来执行 CGI 程序，并将执行结果发送给客户端浏览器。Boa 的功能实现也是通过建立连接、绑定端口、进行侦听、请求处理等来实现的。其初始化部分的源代码如下：

```
int server_s;
server_s = socket( SERVER_PF,SOCK_STREAM,IPPROTO_TCP );
if( server_s == - 1 ) {
    DIE( unable to create socket );
}
if( set_nonblock_fd( server_s ) == - 1 ) {
    DIE( unable to set server socket to nonblocking );
}
if( fcntl( server_s,F_SETFD,1 ) == - 1 ) {
    DIE( can't set close! on! exec on server socket! );
}
if( ( setsockopt( server_s, SOL_SOCKET,SO_REUSEADDR,( void*)&sock_opt,
sizeof( sock_opt ) ) ) == - 1 ) {
    DIE( setsockopt );
}
if( bind_server( server_s, server_ip, server_port ) == - 1 ) {
    DIE( unable to bind );
}
if( listen( server_s, backlog ) == - 1 ) {
    DIE( unable to listen );
}
```


上述代码主要用于打开一个有效的 socket 描述符，然后将其转换为无阻塞套接字。函数 `bind()` 用于建立套接字描述符与指定端口间的关联，并通过函数 `listen()` 在该指定端口侦听，等待远程连接请求。当侦听到连接请求时，Boa 调用函数 `get_request(int server_sock)` 获取请求信息，通过调用函数 `accept()` 为该请求建立一个连接。在建立连接之后，接收请求信息，同时对请求进行分析。当有 CGI 请求时，为 CGI 程序创建进程，并将结果通过管道发送输出。Boa 的整体工作流程如图 8-11 所示。

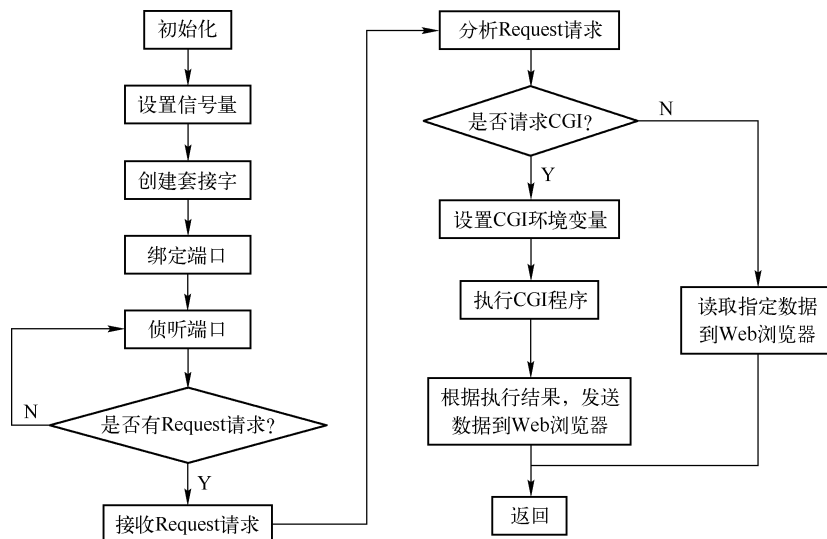


图 8-11 Boa 的工作流程

第二步完成 Boa 的配置，使其能够支持 CGI 程序的执行。Boa 需要在 `/etc` 目录下建立一个 `boa` 目录，里面放入 Boa 的主要配置文件 `boa.conf`。在 Boa 源码目录下已有一个示例 `boa.conf`，可以在其基础上进行修改，下面解释一下该文件的含义。

```

#监听的端口号，默认值是 80，一般无须修改
Port 80
# bind 调用的 IP 地址，一般注释掉，表明绑定到 INADDR_ANY，通配于服务器的所有 IP 地址
#Listen 192.68.0.5
#作为哪个用户运行，即它拥有该用户的权限，一般都是 nobody，需要/etc/passwd 中有 nobody
#用户
User nobody
#作为哪个用户组运行，即它拥有该用户组的权限，一般都是 nogroup，需要在/etc/group 文
#件中有 nogroup 组
Group nogroup
#当服务器发生问题时发送报警的 E-mail 地址，目前未用，注释掉
#ServerAdmin root@localhost
#错误日志文件。如果没有以/开始，则表示从服务器的根路径开始。如果不需要错误日志，则用
#/dev/null。在下面设置时，注意一定要建立/var/log/boa 目录
ErrorLog /var/log/boa/error_log
#访问日志文件。如果没有以/开始，则表示从服务器的根路径开始。如果不需要错误日志，则用

```

```
#/dev/null 或直接注释掉。在下面设置时，注意一定要建立/var/log/boa 目录
#AccessLog /var/log/boa/access_log
#是否使用本地时间。如果没有注释掉，则使用本地时间；注释掉则使用 UTC 时间
#UseLocaltime
#是否记录 CGI 运行信息，如果没有注释掉，则记录；注释掉则不记录
#VerboseCGILogs
#服务器名字
ServerName www.hyesco.com
#是否启动虚拟主机功能，即设备可以有多个网络接口，每个接口都可以拥有一个虚拟的 Web 服务器。一般注释掉，即不需要启动
#VirtualHost
#非常重要，HTML 文档的主目录。如果没有以/开始，则表示从服务器的根路径开始
DocumentRoot /var/www
#如果收到一个用户请求，在用户主目录后再增加的目录名
UserDir public_html
#HTML 目录索引的文件名，也是没有用户只指明访问目录时返回的文件名
DirectoryIndex index.html
#当 HTML 目录没有索引文件时，用户只指明访问目录，Boa 会调用该程序生成索引文件，然后返回给用户，因为该过程比较慢故最好不执行，可以注释掉或者给每个 HTML 目录加上
#DirectoryIndex 指明的文件
#DirectoryMaker /usr/lib/boa/boa_indexer
#如果 DirectoryIndex 不存在，并且 DirectoryMaker 被注释掉，那么就用 Boa 自带的索引生成程序来生成目录的索引文件并输出到下面目录，该目录必须是 Boa 能读写的
#DirectoryCache /var/spool/boa/dircache
#一个连接所允许的 HTTP 持续作用请求最大数目，注释掉或设为 0 都将关闭 HTTP 持续作用
KeepAliveMax 1000
#HTTP 持续作用中服务器在两次请求之间等待的时间数，以秒为单位，超时将关闭连接
KeepAliveTimeout 10
#指明 mime.types 文件位置。如果没有以/开始，则表示从服务器的根路径开始。可以注释掉以避免使用 mime.types 文件，此时需要用 AddType 在本文件中指明
MimeType /etc/mime.types
#文件扩展名没有或未知，使用默认的 MIME 类型
DefaultType text/plain
#提供 CGI 程序的 PATH 环境变量值
CGIPath /bin:/usr/bin:/usr/local/bin
#将文件扩展名和 MIME 类型关联起来，和 mime.types 文件的作用一样。如果用 mime.types 文件，则注释掉；如果不用 mime.types 文件，则必须使用
#AddType application/x-httpd-cgi cgi
#指明文档重定向路径
#Redirect /bar http://elsewhere/feh/bar
#为路径加上别名
Alias /doc /usr/doc
#非常重要，指明 CGI 脚本的虚拟路径对应的实际路径。一般所有的 CGI 脚本都要放在实际路径中，用户访问执行时输入站点+虚拟路径+CGI 脚本名
ScriptAlias /cgi-bin//var/www/cgi-bin/
```

用户可以根据自己的需要对 `boa.conf` 进行修改，但必须保证其他的辅助文件和设置与 `boa.conf` 中的配置相符，否则 `Boa` 就不能正常工作。在上面的例子中，还需要创建日志文件所在目录 `/var/log/boa`，创建 HTML 文档的主目录 `/var/www`，将 `mime.types` 文件复制到 `/etc` 目录下，创建 CGI 脚本所在目录 `/var/www/cgi-bin/`。`mime.types` 文件用来指明不同文件扩展名对应的 MIME 类型，一般可以直接从 Linux 主机上复制一个，大部分也都是在主机的 `/etc` 目录下。

8.7.2 Boa 的移植步骤

本节讲解 `Boa` 的移植，首先在 x86 上试验运行，然后在 ARM 上运行。

下载 `Boa` 的源代码 `boa-0.94.13`，解压后进入 `src` 目录，执行 `./configure` 生成 `Makefile` 文件，`make clean`，然后运行 `make`，便生成了 x86 下的 `Boa` 程序。运行 `./boa` 出现错误，是 `boa.conf` 没有配置好，这时需要对此文件进行设置。另外，CGI 脚本测试很容易发生权限不够的错误，要保证 `Boa` 访问的主目录、CGI 脚本目录及临时文件目录（如果没有设置 `TMP` 环境变量，则默认是 `/tmp` 目录）都必须能被 `Boa` 运行时所代表的用户完全访问，该用户由 `boa.conf` 中的 `User` 指出。

以下为运行 `Boa` 出现的错误及解决方法。然后修改 `/tmp/boa.conf` 服务端口为 8080。

```
[root@localhost src]# ./boa -c .
Could not open mime.types file, "/tmp/web/mime.types", for reading
[root@localhost src]# ./boa
Could not open boa.conf for reading.
[root@localhost src]# mv boa.conf /tmp/web
[root@localhost src]# ./boa
Could not open mime.types file, "/tmp/web/mime.types", for reading
[root@localhost src]# cp ../mime.types /tmp/web
[root@localhost src]# ./boa
[root@localhost src]# _

# Port: The port Boa runs on. The default port for http servers is 80.
# If it is less than 1024, the server must be started as root.

Port 8080

# Listen: the Internet address to bind(2) to. If you leave it out,
# it takes the behavior of port 0.0.0.0, which is to bind to all
```

等 `Boa` 正常运行以后，将 `index.html` 复制到 `/tmp/web` 下，最后测试，在客户端浏览器运行 `http://serverip:8080`。

接下来进行 CGI 脚本的测试，需要一个测试用的 CGI 脚本。可以写一个最简单的“Hello World”程序，示例代码如下：

```
#include <stdio.h>
void main() {
    printf("Content-type: text/html\n\n");
    printf("<html>\n");
    printf("<head><title>CGI Output</title></head>\n");
```

```

    printf("<body>\n");
    printf("<h1>Hello, world.</h1>\n");
    printf("</body>\n");
    printf("</html>\n");
    exit(0);
}

```

将以上文件保存为 `cgitest.c`，先在 x86 上测试，用 `gcc` 编译，`gcc -o cgitest cgitest.c`，然后放在 `/var/www/cgi-bin/` 下面。启动 `Boa`，假设 `Boa` 的配置文件是 `boa.conf`，并且放在 `Boa` 的当前目录，执行“`boa -c`”。输入“`ps -A`”，看看是否有 `Boa` 这个进程。

然后在 IE 中输入“`127.0.0.1/cgi-bin/cgitest`”，应该出现网页，显示“`hello, world`”。

注意：在 x86 上测试时应关闭 Linux 的防火墙。

在 x86 上测试通过以后，下面开始移植到 ARM 上。

首先修改刚才的 `Makefile` 文件，将 `CC` 和 `CPP` 修改为 `arm-linux-gcc`：

```

YACC = bison -y
LEX = flex
CC = arm-linux-gcc
CPP = arm-linux-gcc -E

```

运行 `make` 生成 `Boa` 文件，下载到开发板，执行 `Boa`，出现如下错误：

```

[root@P... /webapplication]$ ./boa
./boa: /lib/libc.so.6: version 'GLIBC_2.2' not found (required by ./boa)

```

说明开发板上的库文件和编译器默认的库文件不一致。经过查找，发现原因是在 Linux 主机上安装了多个版本的 `arm-linux-gcc` 所致，需要采用正确的版本才行。

重新解压交叉编译文件到 `/usr/local` 目录下，然后设置 `PATH` `export PATH=$PATH:/usr/local/arm-linux/bin`，同时除掉不需要的其他交叉工具路径。重新 `make` 并复制到开发板，这样就可以了。

发现 `Boa` 文件比较大，再次执行 `arm-linux-strip boa`，新生成的 `Boa` 文件只有 65KB，适合嵌入式系统。

总结问题：发现交叉编译工具链必须安装在指定的路径，否则无法编译通过，会出现 `cpp0` 找不到的问题，这是因为默认的库找不到。

在网页的 `POST` 命令中添加所提交的 CGI 脚本，发现无法找到，但直接在 IE 中输入又正常。

查看 `error_log`，发现“`...mkstemp: Permission denied`”。浏览器弹出错误对话框，“此文档中无数据”。

原因是在 `Boa` 源代码目录下的 `util.c` 文件中，用 `mkstemp()` 函数创建一个临时文件时出现权限错误。这是由于在 `boa.conf` 文件中设置了“`user: nobody`”，使其运行 `Boa` 服务器时以 `nobody` 为用户（可以用 `ps` 命令查看），所以在创建临时文件时没有足够的权限，可以在 `boa.conf` 中将运行 `Boa` 的用户设为 `root` 身份（`user: root`）。

当设置完 “user: root”，运行 Boa 时，在终端会输出错误信息：boa.c:266.icky Linux kernel bug!No such file。原因是 boa.c 文件中的语句出现问题，可以将 boa.c 文件中的第 266 行屏蔽掉。

C 语言并不是很适合开发像 CGI 这种需要大量进行字符串操作的程序，编程比较烦琐，因此，对于一个专业的开发人员来说，首先想到的应该是有没有可复用的库来支持快速高效地开发 CGI 程序。幸运的是目前就有不少开放源码的支持 CGI 开发的 C 库。我们在此只介绍 CGIC，有兴趣的朋友可以自己在 Internet 上搜索其他的 C 库。

8.7.3 CGIC 库的移植

CGIC 是一个支持 CGI 开发的开放源码的标准 C 库，可以免费使用，只需在开发的站点和程序文档中有一个公开声明即可，表明程序使用了 CGIC 库。用户也可以购买商业授权而无须进行公开声明。

CGIC 能够提供以下功能：

- (1) 分析数据，并自动校正一些有缺陷的浏览器发来的数据；
- (2) 透明接收用 GET 或 POST 方法发来的 From 数据；
- (3) 能接收上传文件；
- (4) 能够设置和接收 cookies；
- (5) 用一致的方式处理 From 元素里的回车；
- (6) 提供字符串、整数、浮点数、单选或多选功能来接收数据；
- (7) 提供数字字段的边界检查；
- (8) 能够将 CGI 环境变量转化成 C 中的非空字符串；
- (9) 提供 CGI 程序的调试手段，能够回放 CGI 程序执行时的 CGI 状态。

总之，CGIC 是一个功能比较强大的支持 CGI 开发的标准 C 库，并支持 Linux、UNIX 和 Windows 等多种操作系统。

以下介绍 CGIC 的移植过程。

从 CGIC 的主站点 <http://www.boutell.com/cgi/> 上下载源码，当前最新版本是 2.05 版。将其解压并进入源码目录：

```
# tar xzf cgic205.tar.gz
# cd cgic205
```

修改 Makefile 文件。找到 CC=gcc，将其改成 CC=arm-linux-gcc；找到 AR=ar，将其改成 AR=arm-linux-ar；找到 RANLIB=ranlib，将其改成 RANLIB=arm-linux-ranlib；找到 gcc cgictest.o -o cgictest.cgi \${LIBS}，将其改成\$(CC) \$(CFLAGS) cgictest.o -o cgictest.cgi \${LIBS}；找到 gcc capture.o -o capture \${LIBS}，将其改成\$(CC) \$(CFLAGS) capture.o -o capture \${LIBS}，并保存退出。

然后运行 make 进行编译，得到 CGIC 库 libcgic.a，通过调试辅助程序 capture 和测试程序 cgictest.cgi 来验证生成 CGIC 库的正确性。

将 capture 和 cgictest.cgi 复制到主机的/nfs/www/cgi-bin 目录下。

在工作站的浏览器地址栏中输入“<http://127.0.0.1/cgi-bin/cgicest.cgi>”，可以看到页面，表示 CGIC 库和测试脚本移植成功。cgicest.cgi 比较完整地展现了 CGIC 库的功能，在开发基于 CGIC 库的 CGI 程序前最好先掌握 cgicest.cgi 程序，它也是用户开发特定应用程序时的参考范例。

8.7.4 HTML 模板的制作

Web 方式的应用开发一般都会将界面和程序逻辑脱离开来，允许在一定程度下更改界面，如改变界面文本的属性、建立多语言版本等，而无须改动程序逻辑。界面一般由美工制作，程序员负责具体功能的实现。在 HTML 中，表单（Form）是最主要的传递信息的手段，它适用于任何浏览器。表单中有很多元素，包括输入文本框、单选框、多选框、按钮，等等，可以提供信息的交互。具体对象说明和语法请参见其他 HTML 书籍，这里不予介绍。根据应用需求，美工或其他设计人员将最后的 Web 页面设计出来，作为程序员进行开发的模板。

CGI 程序的工作一般是接收表单数据，进行数据处理，最后根据处理结果生成新的页面返回给浏览器。表单数据通常是以 POST 方法提交给服务器的，由 CGI 程序获得，程序必须将界面数据和内部数据对应起来才能进行下一步的处理。CGI 程序从页面获取数据就根据元素名称/值中的元素名字来进行区分。但 CGI 返回页面比较麻烦。由于界面在程序开发完成后还有可能会改变，而且有些需要程序处理的地方可能没有表单元素，因此对程序来说，不能以表单元素名作为区分的基础，一般的方法是采用 HTML 中的注释 `<!--xxx-->` 来标记。程序员需要在模板中为每一个表单元素在任何需要程序处理的地方，按照一定规则，如注释的下一行就是表单元素行，建立其注释标记。CGI 程序可以根据注释标记来判断表单元素信息并进行处理。程序逐行读取模板文件，检查有无注释标记，如果有，则下一行需要进行处理，给表单元素赋上数据，最后就可以返回带数据的页面给浏览器了。

HTML 模板还需要关注输入的检查。根据输入检查越早越好的原则，需要在用户界面上对用户提交的数据进行检查。目前一般是采用 JavaScript 脚本的方式。当用户提交数据时，表单对象的 `onSubmit` 方法会被调用，在该方法里可以对用户的输入进行检查。常用的检查有是否必需、最大/小长度、是否为字符、是否为数字、E-mail 地址是否正确、IP 地址是否正确、是否匹配一个正则表达式等。

为了更加清楚地了解 GET 和 POST 方法，读者可以先把 Boa 压缩包中的 `cgi-test.cgi` 复制到 Boa 的 CGI 目录下，然后在 IE 中输入图 8-12 所示内容。

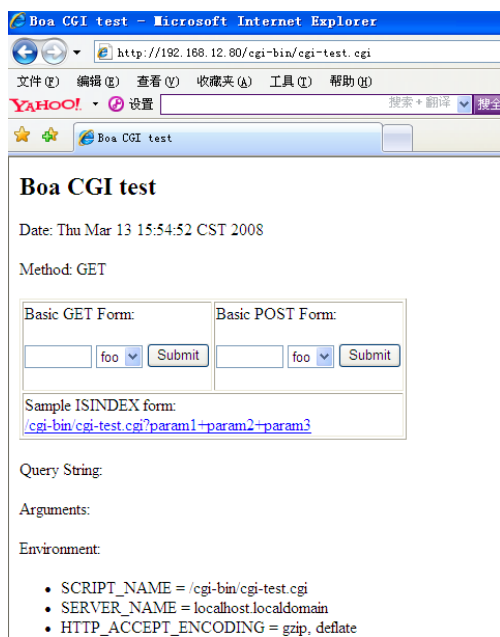


图 8-12 CGI 测试页面

用户可以自己在 GET 和 POST 表格内输入测试, 仔细观察显示结果。

8.7.5 一个综合的 Web 测试实验

下面以家庭网关的用户登录界面为例, 来实现一个 JavaScript+CGI+Boa 的试验。

如图 8-13 所示, 当输入用户名和密码后, CGI 脚本把处理结果返回给 IE 浏览器。测试网页放在 Boa 服务器上, CGI 也放在 boa 服务器上, 当网页内的 POST 提交后, CGI 接收到参数, 解析后返回给 IE。

在浏览器中出现:

图 8-13 登录界面

```
用户名  xxxx
密码   xxxxx
```

首先要研究如何将输入的用户名和密码发送到远程, 这里采用的是表单技术。先来看以上网页的代码:

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312" />
<title>无标题文档</title>
</head>

<body>
<table width="800" height="400" bgcolor="#FFFF33">
  <!--DWLayoutTable-->
  <tr>
    <td width="790" height="594"><table width="788" height="432">
      <tr>
        <td><div align="center"><span class="STYLE1">家庭网关登录</span></div></td>
      </tr>
      <tr>
        <td><form id="form1" name="form1" method="post" action="cgi-bin/usrlogin.cgi">
          <label></label>
          <label></label>
          <p align="center">
            <input name="login_command" type="hidden" id="login_command" />
            <input name="table_name" type="hidden" id="table_name" value="user_table" />
          </p>
          <p align="center">用户名
            <input name="username" type="text" id="username" />
          </p>
          <p align="center">密 码
            <input name="password" type="password" id="password" />
          </td>
        </td>
      </tr>
    </table>
  </tr>
</table>
```

```

        </p>
        <p align="center">
            <input name="confirm_login" type="submit" id="confirm_login" value="登录" />
        </p>
        <p>&nbsp;</p>
    </form></td>
        </tr>
    </table></td>
</tr>
</table>
<p align="center" class="STYLE1">&nbsp;</p>
</body>
</html>

```

在以上<form>、</form>之间的是表单，参数就放在其中。

```
<form id="form1" name="form1" method="post" action="cgi-bin/usrlogin.cgi">
```

表示当提交的时候，采用 POST 方法发送参数，并请求服务器上的 cgi-bin/usrlogin.cgi 程序来处理提交的参数。

```

    <p align="center">
        <input name="login_command" type="hidden" id="login_command" />
        <input name="table_name" type="hidden" id="table_name" value="user_table" />
    </p>

```

以上两行表示提交的参数在网页中看不到，如果需要提交的参数不想让用户看到，可以这么做。

```

<p align="center">用户名
    <input name="username" type="text" id="username" />
</p>
<p align="center">密 码
    <input name="password" type="password" id="password" />
</p>

```

这里就是提交的“名称/值”对。我们提交的有“username/xxx”、“passwd/xxx”4个参数。

```

<p align="center">
    <input name="confirm_login" type="submit" id="confirm_login" value="登录" />

```

上面这句表示提交，当按“登录”按钮以后，就发送以上“名称/值”对，一共是4个参数，其中两个隐藏参数。

POST 方法发送到服务器后，需要解析出提交的“名称/值”对。在 Boa 中是采用 char **getcgivars()函数来提取的。

其原型如下（在文件 httpcgiparse.c 中）：

```
char **getcgivars() {
```



```

register int i ;
char *request_method ;
int content_length;
char *cgiinput ;
char **cgivars ;
char **pairlist ;
int paircount ;
char *nvpair ;
char *eqpos ;

/** Depending on the request method, read all CGI input into cgiinput. */
request_method= getenv("REQUEST_METHOD") ;
//从客户端传递来的请求中分析是 GET 还是 POST 方法
if (!strcmp(request_method, "GET") || !strcmp(request_method, "HEAD")) {
    /* Some servers apparently don't provide QUERY_STRING if it's empty, */
    /* so avoid strdup()'ing a NULL pointer here. */
    char *qs ;
    qs= getenv("QUERY_STRING") ;
//将附在 URL 的 GET 后的查询字符串取出
    cgiinput= strdup(qs ? qs : "");
//将 qs 复制到 cgiinput 指向的字符串
}
//以下是 POST 方法
else if (!strcmp(request_method, "POST")) {
    /* strcasecmp() is not supported in Windows-- use strcmpi() instead */
    if ( !strcasecmp(getenv("CONTENT_TYPE"), "application/x-www-form-urlencoded")) {
        printf("Content-Type: text/plain\n\n") ;
        printf("getcgivars(): Unsupported Content-Type.\n") ;
        exit(1) ;
    }
    if ( !(content_length = atoi(getenv("CONTENT_LENGTH"))) ) {
        printf("Content-Type: text/plain\n\n") ;
        printf("getcgivars(): No Content-Length was sent with the POST request.\n") ;
        exit(1) ;
    }
    if ( !(cgiinput= (char *) malloc(content_length+1)) ) {
        printf("Content-Type: text/plain\n\n") ;
        printf("getcgivars(): Couldn't malloc for cgiinput.\n") ;
        exit(1) ;
    }
//从 stdin 读取 CGI 输入变量
    if (!fread(cgiinput, content_length, 1, stdin)) {
        printf("Content-Type: text/plain\n\n") ;
        printf("getcgivars(): Couldn't read CGI input from STDIN.\n") ;
        exit(1) ;
    }
}

```

```

    }
    cgiinput[content_length]='\0';
}
else {
    printf("Content-Type: text/plain\n\n");
    printf("getcgivars(): Unsupported REQUEST_METHOD.\n");
    exit(1);
}

/** Change all plusses back to spaces. */
for (i=0; cgiinput[i]; i++) if (cgiinput[i] == '+') cgiinput[i] = ' ';

/** First, split on "&" and ";" to extract the name-value pairs into */
/** pairlist.                                                                    */
pairlist= (char **) malloc(256*sizeof(char **));
paircount= 0;
nvpair= strtok(cgiinput, "&");

```

strtok 表示分解字符串为一组标记串，cgiinput 为要分解的字符串，& 为分隔符字符串。strtok 在 cgiinput 中查找包含在 & 中的字符并用 NULL("\0")来替换，直到找遍整个字符串。函数返回指向下一个标记串。

```

while (nvpair) {
    //如果指向的下一个分解的字符串不为空，将字符串复制到 pairlist 中
    pairlist[paircount++] = strdup(nvpair);
    if (!(paircount%256))
        pairlist= (char **) realloc(pairlist, (paircount+256)*sizeof(char **));
    nvpair= strtok(NULL, "&");
}
pairlist[paircount]= 0;    /* terminate the list with NULL */

/** Then, from the list of pairs, extract the names and values. */
在 pairlist 中取出“名字/值”对：
cgivars= (char **) malloc((paircount*2+1)*sizeof(char **));
for (i= 0; i<paircount; i++) {
    if (eqpos=strchr(pairlist[i], '=')) {

```

查找字符串 pairlist[i] 中首次出现字符=的位置，返回首次出现=的位置的指针，如果不存在 = 则返回 NULL。

```

//提取=右边的值，放入 cgivars[i*2+1]
    *eqpos= '\0';
    unescape_url(cgivars[i*2+1]= strdup(eqpos+1));
} else {
    unescape_url(cgivars[i*2+1]= strdup(""));
}

```

```

        unescape_url(cgivars[i*2]= strdup(pairlist[i]));
    }
    cgivars[paircount*2]= 0;    /* terminate the list with NULL */
    //提取=左边的名字，放入 cgivars[i*2]
    /** Free anything that needs to be freed. */
    free(cgiinput);
    for (i=0; pairlist[i]; i++) free(pairlist[i]);
    free(pairlist);

    /** Return the list of name-value strings. */
    return cgivars;
    //返回的 cgivar 字符串指针指向“名字/值”对数组
};

```

cgivars 存放了提取 POST 提交的数据，下面的 CGI 程序负责向 IE 回送接收到的“名字/值”对：

```

/** Standard "hello, world" program, that also shows all CGI input. */
int main() {

    char **cgivars;
    int i;

    /** First, get the CGI variables into a list of strings */
    cgivars= getcgivars();

    /** Print the CGI response header, required for all HTML output. */
    /** Note the extra \n, to send the blank line. */
    printf("Content-type: text/html\n\n");

    /** Finally, print out the complete HTML response page. */
    printf("<html>\n");
    printf("<head><title>CGI Results</title></head>\n");
    printf("<body>\n");
    printf("<h1>Hello, world.</h1>\n");
    printf("Your CGI input variables were:\n");
    printf("<ul>\n");

    /** Print the CGI variables sent by the user. Note the list of */
    /** variables alternates names and values, and ends in NULL. */
    for (i=0; cgivars[i]; i+= 2)
        printf("<li>[%s] = [%s]\n", cgivars[i], cgivars[i+1]);
    printf("</ul>\n");
    printf("</body>\n");
    printf("</html>\n");

    /** Free anything that needs to be freed */
    for (i=0; cgivars[i]; i++) free(cgivars[i]);
    free(cgivars);
}

```

```

    exit(0);
}

```

将以上文件命名为 `cgitest1.c`，首先在 x86 上测试。
运行：

```

gcc -c cgitest1.c
gcc -c httpcgiparse.c
gcc -o cgitest1 cgitest1.o httpcgiparse.o

```

将程序编译为 `cgitest1.cgi`。

为了和网页的 POST 方法指定的 CGI 文件名一致，这里改名为 `usrlogin.cgi`。

将家庭网关登录网页改名为 `index.html`，放入 `/var/www/` 目录中。

假设服务器是 192.168.12.80，在 IE 中输入网址出现登录界面，用户名输入“abc”，密码输入“123456”，单击“登录”按钮，出现如图 8-14 所示结果。

其中前面两项是隐藏的提交值，`username` 和 `passwd` 是刚才输入的数据。

以上代码是直接打印输出数据，而一般的网页在提交后希望网页页面不变，在其中规定的区域显示提交返回的结果。因此，需要通过 JavaScript 来接收服务器返回的参数，并且在网页上显示出来。

现在研究一下 IP 设置页面的 JavaScript 功能，IP 设置的界面如图 8-15 所示。



图 8-14 Web 返回查询结果

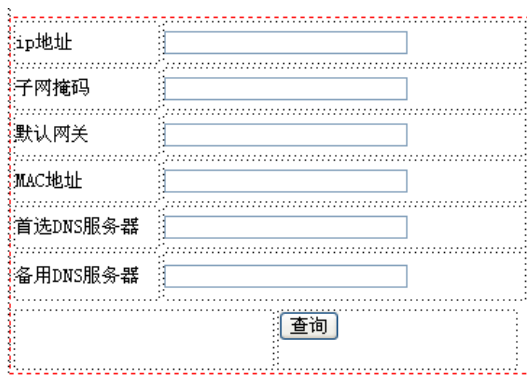


图 8-15 IP 设置页面的 JavaScript 功能

当按“查询”按钮后，显示服务器返回的 IP 地址设置。

网页代码如下：

```

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>无标题文档</title>
</head>

```

```

<body onLoad="init()">

<table width="395" border="0" cellpadding="0" cellspacing="0">
  <tr>
    <td width="360" height="386">
      <td><form id="form1" name="form1" method="post" action="/cgi-bin/ipsetting.cgi">
        <table width="360" height="200">
          <tr>
            <td width="99">ip 地址</td>
            <td width="249"><input name="ipaddr" type="text" id="ipaddr" onKeyDown="check
(value)" value=" "></td>
          </tr>
          <tr>
            <td>子网掩码</td>
            <td><input name="ipmask" type="text" id="ipmask" value=" "></td>
          </tr>
          <tr>
            <td>默认网关</td>
            <td><input name="gateway" type="text" id="gateway" value=" "></td>
          </tr>
          <tr>
            <td>MAC 地址</td>
            <td width="249"><input name="macaddr" type="text" id="macaddr" onKeyDown="check
(value)" value=" "></td>
          </tr>
          <tr>
            <td>首选 DNS 服务器</td>
            <td><input name="dns1" type="text" id="dns1" value=" "></td>
          </tr>
          <tr>
            <td>备用 DNS 服务器</td>
            <td><input name="dns2" type="text" id="dns2" value=" "></td>
          </tr>
        </table>
        <table width="354" height="48">
          <tr>
            <td width="178" height="42">&nbsp;</td>
            <td width="164"><p>
              <input name="Submit" type="submit" value="查询">
            </p>
            <p></td>
          </tr>
        </table>
      </td>
    </tr>
  </table>

```

```
</table>
</form>
<div align="left"></div></td>
</tr>
</table>
</body>
</html>
<script language="javascript">
var injs=new Array(10);

function check(ip)
{
}
function init()
{
form1.ipaddr.value=injs[0];
form1.ipmask.value=injs[1];
form1.gateway.value=injs[2];
form1.macaddr.value=injs[3];
form1.dns1.value=injs[4];
form1.dns2.value=injs[5];
}
</script>
```

网页中有一个表单 form1，其中有 ipaddr、ipmask、gateway、macaddr、dns1、dns2 这些常规 IP 设置。

```
<body onload="init()">
```

上述语句用于告诉 IE 在显示网页前先运行 init 这个函数。

```
<script language="javascript">
```

上述语句表示采用 JavaScript 脚本。

function init()表示函数名是 init。

```
function init()
{
form1.ipaddr.value=injs[0];
form1.ipmask.value=injs[1];
form1.gateway.value=injs[2];
form1.macaddr.value=injs[3];
form1.dns1.value=injs[4];
form1.dns2.value=injs[5];
}
</script>
```

以上表示从接收到的 injs 字符串数组中取出数据, injs 是从嵌入式服务器发送过来的。为直观了解以上代码的工作方式, 这里把 injs 数据写在 JavaScript 下面来模拟服务器发来的网页。

```
function init()
{
    form1.ipaddr.value=injs[0];
    form1.ipmask.value=injs[1];
    form1.gateway.value=injs[2];
    form1.macaddr.value=injs[3];
    form1.dns1.value=injs[4];
    form1.dns2.value=injs[5];
}
</script>
<script language="javascript">
    injs[0]='192.168.1.12';injs[1]='255.255.255.0';injs[2]='192.168.1.1';injs[3]='11:22:33:44:55:66';injs[4]='202.120.224.6';injs[5]='202.120.224.104';
</script>
```

ip地址	<input type="text" value="192.168.1.12"/>
子网掩码	<input type="text" value="255.255.255.0"/>
默认网关	<input type="text" value="192.168.1.1"/>
MAC地址	<input type="text" value="11:22:33:44:55:66"/>
首选DNS服务器	<input type="text" value="202.120.224.6"/>
备用DNS服务器	<input type="text" value="202.120.224.104"/>
<input type="button" value="查询"/>	

图 8-16 IE 收到参数后显示

以上加粗的代码就是模拟服务器返回的数据, 它直接附在网页的末尾。

将以上文件命名为 ipconfig-javascript.html, 用 IE 在本地打开, 可以看到数据, 如图 8-16 所示。

以上只是在网页模拟服务器返回的数据, 这些数据放在<script language="javascript"> </script>中间, 可以被 JavaScript 程序读到。接着看嵌入式 CGI 是如何真正返回数据的。

首先, CGI 读取整个网页文件并发送到远程, 这样, 用户在 IE 中直接输入 “http://192.168.12.80/xxx.cgi” 就能读取对应的网页。接着在 CGI 程序的末尾, 有如下代码行:

```
printf("%s", "<script language=javascript>");
printf("%s", outjs);
printf("%s", "</script>");
```

outjs 存放的是字符串。

```
outjs=<script language="javascript">
    injs[0]='192.168.1.12';injs[1]='255.255.255.0';injs[2]='192.168.1.1';injs[3]='11:22:33:44:55:66';injs[4]='202.120.224.6';injs[5]='202.120.224.104';
</script>
```

这样, 就把参数发给 IE 了。

假设以上代码是 cgitest2.c。将 Ipconfig.html 中的 POST 部分修改为 cgitest2.cgi。

将 ipconfig.html 复制到 var/www，将编译好的 cgitest2.cgi 复制到/var/www/cgi-bin/。运行程序，如图 8-17 所示。

真正的 CGI 返回的数据被显示，如图 8-18 所示。

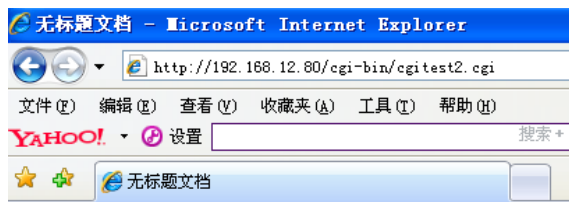


图 8-17 运行程序

ip地址	<input type="text" value="192.168.1.12"/>
子网掩码	<input type="text" value="255.255.255.0"/>
默认网关	<input type="text" value="192.168.1.1"/>
MAC地址	<input type="text" value="11:22:33:44:55:66"/>
首选DNS服务器	<input type="text" value="202.120.224.6"/>
备用DNS服务器	<input type="text" value="202.120.224.104"/>
<input type="button" value="查询"/>	

图 8-18 真正的 CGI 返回的数据被显示

以上 CGI 输出的是本地 IP 信息，但不是从系统运行环境中获取的，是我们任意给定的。下面就需要从系统环境中取出了。首先，研究最简单的例子，在程序中读取 Linux 的 MAC 地址，并在屏幕上输出。

程序是 macaddr.c。

```
#include <stdio.h>
#include <stdlib.h>
#include <asm/ioctl.h>
#include <asm/errno.h>
#include <linux/slab.h>
#include <fcntl.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>

main()
{
    unsigned char mac_str[]="ifconfig eth0 |grep HWaddr |cut -dr -f3 |cut -d\\ -f2 > /tmp/mac_addr0";
    unsigned char ip_str[] = "ifconfig eth0 |grep inet |cut -d: -f2 |cut -d\\ -f1 > /tmp/ip_addr0";
    unsigned char net_str[]="ifconfig eth0 |grep Mask |cut -dk -f2 |cut -d: -f2 > /tmp/net_addr0";
    unsigned char gw_str[]="route |grep default |cut -dt -f2 |cut -d0 -f1 |cut -d\\ -f10 > /tmp/gw_addr0";
    unsigned char    mac_addr[24];
    unsigned char    ip_addr[24];
    unsigned char    net_addr[24];
    unsigned char    gw_addr[24];
    unsigned char mac_sys[80];
    int error_sys = 0;
    FILE *mac_fp, *ip_fp, *gw_fp, *net_fp;
    /*Creat temp mac_addr file*/
```



```

strcpy(mac_sys, "\\0");
strcat(mac_sys, mac_str);
if((error_sys = system(mac_sys)) != 0)
{
    //      DEV_LEVEL1("[set_eth] mac_sys : 0x%x\\n", error_sys);
}
if ((mac_fp=fopen("/tmp/mac_addr", "r")) != NULL) {
    fread(mac_addr, 1, 18, mac_fp);
}
else {
    perror ("fread");
    return 0;
}
fclose (mac_fp);
printf("### MAC address : %s\\n", mac_addr);
}

```

其中，看下面一句：

```
unsigned char mac_str[]="ifconfig eth0 |grep HWaddr |cut -dr -f3 |cut -d\\ -f2 > /tmp/mac_addr\\0";
```

ifconfig eth0 |grep HWaddr 是从 ifconfig eth0 输出结果中提取 Hwaddr 这一行，cut -dr -f3 是将字符 r 作为分隔符，取出第三个字符串，这里就是具体的 mac 地址数据。

system(mac_sys)表示将 mac_sys 提交给 Linux 执行系统调用。将 mac 地址存入文件 /tmp/mac_addr 中。

```

if ((mac_fp=fopen("/tmp/mac_addr", "r")) != NULL) {
    fread(mac_addr, 1, 18, mac_fp);
}

```

是从/tmp/mac_addr 中取出 mac 地址，放在字符串 mac_addr 中。

将以上程序用 arm-linux-gcc-o macaddr macaddr.c 下载到开发板运行，观察结果。

ipaddr.c 程序演示完整的提取本地 IP 信息过程。

ipset.c 是在程序中设置 IP 地址。

```

#include <stdlib.h>
#include <asm/ioctl.h>
#include <asm/errno.h>
#include <linux/slab.h>
#include <fcntl.h>
#include <pthread.h>
#include <string.h>
#include <unistd.h>
typedef struct __SYS_CONF__{
    unsigned char  host_name[24];

```

```

unsigned char  mac_addr[24];

unsigned char  ip_addr[4];
unsigned char  net_mask[4];
unsigned char  gw_addr[4];

}SYS_CONF;

SYS_CONF sys_conf_str;
int main(int argc, char *argv[])
{
    char s_sys[80], ifconfig_sys[80] = "ifconfig", route_sys[80] = "route";
    char hostname_sys[80] = "hostname";
    char host_name[24];
    char ip1_sys[20];
    char netmask1_sys[20];
    char gw1_sys[20];
    int error_eth = 0;
    int i, k;
    //以下是在程序中设置 IP 地址和 netmask、gateway
    sys_conf_str.ip_addr[0]=192;
    sys_conf_str.ip_addr[1]=168;
    sys_conf_str.ip_addr[2]=12;
    sys_conf_str.ip_addr[3]=19;
    sys_conf_str.gw_addr[0]=192;
    sys_conf_str.gw_addr[1]=168;
    sys_conf_str.gw_addr[2]=12;
    sys_conf_str.gw_addr[3]=122;
    sys_conf_str.net_mask[0]=255;
    sys_conf_str.net_mask[1]=255;
    sys_conf_str.net_mask[2]=255;
    sys_conf_str.net_mask[3]=0;
    strcpy(sys_conf_str.host_name, "www.armv.cn");
    printf("[cfg_eth]\n");
    if ((sys_conf_str.ip_addr[0]<=0) || (sys_conf_str.ip_addr[0]>=255) ) {
        printf ("[cfg_eth]Invalid IP address\n");
        return 1;
    }
    if ((sys_conf_str.gw_addr[0]<=0) || (sys_conf_str.gw_addr[0]>=255) ) {
        printf ("[cfg_eth]Invalid gateway address\n");
        return 1;
    }

    strcpy(ip1_sys, "\0"); //strcat(ip1_sys, sys_conf_str.ip_addr);
    sprintf(ip1_sys, "%d.%d.%d.%d",

```

```

        sys_conf_str.ip_addr[0]&0xFF, sys_conf_str.ip_addr[1]&0xFF,
        sys_conf_str.ip_addr[2]&0xFF, sys_conf_str.ip_addr[3]&0xFF);

strcpy(netmask1_sys, "\0");//strcat(netmask1_sys, sys_conf_str.net_mask);
sprintf(netmask1_sys, "%d.%d.%d.%d",
        sys_conf_str.net_mask[0]&0xFF, sys_conf_str.net_mask[1]&0xFF,
        sys_conf_str.net_mask[2]&0xFF, sys_conf_str.net_mask[3]&0xFF);

strcat(ifconfig_sys, " eth0 ");
strcat(ifconfig_sys, ip1_sys);
strcat(ifconfig_sys, " netmask ");
strcat(ifconfig_sys, netmask1_sys);
strcat(ifconfig_sys, " up");
if((error_eth = system(ifconfig_sys)) !=0)
{
    printf("[cfg_eth] ifconfig_sys : 0x%x\n", error_eth);
}
printf ("%s\n", ifconfig_sys);

strcpy(gw1_sys, "\0");//strcat(gw1_sys, sys_conf_str.gw_addr);
sprintf(gw1_sys, "%d.%d.%d.%d",
        sys_conf_str.gw_addr[0]&0xFF, sys_conf_str.gw_addr[1]&0xFF,
        sys_conf_str.gw_addr[2]&0xFF, sys_conf_str.gw_addr[3]&0xFF);

strcat(route_sys, " add default gw ");
strcat(route_sys, gw1_sys);
strcat(route_sys, " eth0");
if((error_eth = system(route_sys)) !=0)
{
    printf("[cfg_eth] route_sys : 0x%x\n", error_eth);
}
printf ("%s\n", route_sys);

strcpy(host_name, "\0");
sprintf(host_name, " %s", sys_conf_str.host_name);
strcat(hostname_sys, host_name);
if((error_eth = system(hostname_sys)) !=0)
{
    printf("[cfg_eth] hostname_sys : 0x%x\n", error_eth);
}
printf ("%s\n", hostname_sys);

return error_eth;
}

```

程序将命令字符串提交 system 执行，将修改开发板的 IP 信息。

将以上程序用 `arm-linux-gcc -o macaddr macaddr.c` 下载到开发板运行，然后在 shell 中输入 `ifconfig eth0` 观察结果。

接着，我们需要在以上两个试验的基础上设计 CGI，从而和网页交互。通过网页可以设置开发板的 IP 地址，也可以读取开发板的当前 IP 地址，如图 8-19 所示。

代码在 `ipsettingtest` 中，可以自动查询当前环境的 IP 信息，并可以修改 IP。

```
Arm-linux-gcc -o ipsettingtest.cgi ipsettingtest.c httpcgiparse.c
```

CGI 将代码编译为 `ipsettingtest.cgi`，放入 Boa 的 CGI 目录，对应的 HTML 文件放入 Boa 的主目录，在浏览器中输入“`http://192.168.x.x/cgi-bin/ipsettingtest.cgi`”，出现如图 8-20 所示界面。

ip地址

子网掩码

默认网关

图 8-19 通过网页设置开发板的 IP 地址

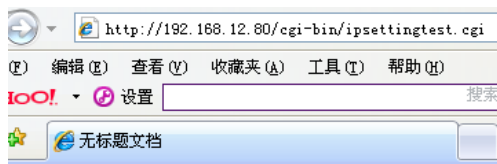


图 8-20 网页界面

尝试修改网关地址为“192.168.12.3”，然后通过开发板的串口终端输入“`route`”，应该出现刚才修改的网关地址。

8.8 通过网络远程控制开发板上的灯

下面进一步在以上试验的基础上，研究如何通过 Web 来控制开发板上的设备动作，具体就是通过网页设置实现某个 LED 的亮灭。

首先，编译 LED 驱动程序。

考虑到开发板已经有驱动程序，为了避免冲突，这里将设备号定义为 68。

编译好的驱动名称为 `XSB_EDR_8LED.o`，复制到开发板，运行命令“`insmod XSB_EDR_8LED.o`”。

然后建立设备节点：

```
mknod "/dev/emdoor_8led_for_boa" c 68 1
```

驱动程序算是建好了。

应用测试程序在\web 服务器教材\cgi\通过网页点灯\basic 目录下。

```
root@localhost module1# arm-linux-gcc -o 8LED_test 8LED_test.c httpcgiparse.c
-I /usr/include/_
```

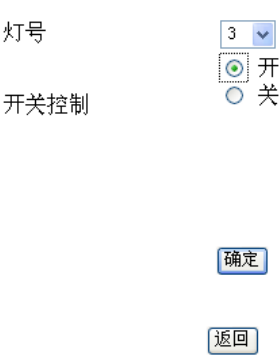


图 8-21 通过网页控制远程开发板的 LED 灯

编译的可执行文件是 8LED_test，将其复制到开发板，运行测试程序。

例如：

```
打开第 0 个灯    8LED_test 0 1
关闭第 0 个灯    8LED_test 0 0
```

接着设计网页，可以控制 8 个 LED 灯。网页界面如图 8-21 所示。

通过在网页选择灯并设置对应的开关状态，开发板上的 LED 应该发生了变化。

以下是网页代码：

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>无标题文档</title>
</head>

<body onLoad="init()">
<table width="390" border="0" cellpadding="0" cellspacing="0">
<tr>
<td width="390" height="382"><form name="form1" method="post" action="/cgi-bin/ledcgi.cgi">

<table width="274">
<tr>
<td>灯号</td>
<td><select name="select">
<option value="2">2</option>
<option value="3">3</option>
<option value="4">4</option>
<option value="5">5</option>
<option value="6">6</option>
<option value="7">7</option>
<option value="8">8</option>
<option value="9">9</option>
<option value="10">10</option>
</select></td>
</tr>
<tr>
<td>开关控制</td>
<td><p>
```

```

        <label>
            <input name="RadioGroup1" type="radio" id="radio0" value="open" >
            开</label>
        <br>
        <label>
            <input name="RadioGroup1" type="radio" id="radio1" value="close" checked>
            关</label>
        <br>
    </p>

    <br>
</td>
</tr>
</table>
<p> <p>

<table width="205" height="35">
    <tr>
        <td width="151"> </td>
        <td width="42"><input type="submit" name="Submit2" value="确定"></td>
    </tr>
</table>
<label></label>
<table width="216" height="83">
    <tr>
        <td width="144"> </td>
        <td width="60">
            </form>
        </td>
    </tr>
</table>
<table width="220" height="44">
    <tr>
        <td width="143"> </td>
        <td width="83"><input type="submit" name="Submit" value="返回" onClick="history.go(-
1)"></td>
    </tr>
</table></td>
</tr>
</table>

</body>

</html>

```

```

<script language="javascript">
var injs=new Array(3);
function init()
{
form1.select.value=injs[0];
if(injs[1]=="open")
form1.RadioGroup1.radio0.checked=true;
else
form1.RadioGroup1.radio0.checked=false;
}
</script>

```

读者可以自己分析，其中新加了 radio button 和 select 控件。

和网页对应的 CGI 程序接收网页提交的数据，也就是当前指定哪个灯亮灭。然后通过 system 调用来运行程序，system 调用的可执行程序可以采用前面测试用的 8LED_test。因此，必须将 8LED_test 文件放在 CGI 可以找到的地方，这里直接复制到 Boa 的 CGI 目录下面。

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv){
    FILE *fp;
    char buffer[5000];
    char **cgivars ;
    int i,j;
    unsigned char ledno[80];
    char comd[80]="8LED_test ";

    cgivars= getcgivars() ;
    fp=fopen("../led.html","r");    //open the file for send to web  browser
    if(fp==NULL)    {
        printf("%s","file does not exist");}
    fread(buffer,sizeof(buffer), 1 , fp);//read web page
    printf("%s",buffer);
    strcat(comd,cgivars[1]);
    strcat(comd," ");
    if(strcmp(cgivars[3],"open"))
        strcat(comd,"0");
    else

```

```
    strcat(cmd,"1");
    system(cmd);
    if(fp!=NULL) fclose(fp);
    return 0;
}
```

以上程序主要是接收 `cgivars[1]`、`cgivars[3]`，其中第一个存放灯的号，第二个表示是 `open` 还是 `close`。由于网页参数发来的是字符串 `open/close`，为了与应用程序配合工作，需要将其转换为字符串 `0` 或 `1`。最后 `system` 调用形成 `8LED_test 21` 这种方式。当然，该程序比较简单，读者也可以自己将一些状态反馈信息加到网页来表示当前开发板设备的开关状态。

以上程序编译完毕放在 `boa.conf` 指定的 CGI 目录下，而网页放在 `boa.conf` 指定的网页目录下。

然后输入“`http://192.168.xx.xx/led.html`”测试，如果没有问题，则开发板的 LED 灯会根据网页而变化。

第 9 章

基于 Linux 的家庭网关设计

前面两章分别学习了 Linux 下的文件系统、嵌入式 Web 和数据库 Sqlite，为本章的学习奠定了基础，本章将重点应用第 8 章的知识，构建一个完整的产品中的网络应用部分。通过本章的学习，读者将能开发远程网络控制应用系统。

9.1 产品开发背景

家庭网关是家庭网络的核心部件,是智能家居的主要部分,通过它可以实现系统信息的采集、信息输入、逻辑处理、信息输出、联动控制等。家庭网关于20世纪80年代兴起于欧美、日本,90年代引进中国。随着我国经济的迅速发展,尤其是近年来房地产业的繁荣,人们对智能家居的概念有了一定的了解,现代人对一个楼盘、别墅评价的重要指标之一,就是家居的智能化程度及安防设施是否齐全。

家庭网关设备是面向家庭用户的智能接入设备,可以它为中心建立家庭网络,并在多个设备间共享Internet网络连接,同时为用户提供安全的通信、娱乐、存储一体化功能。作为家庭网络的核心,家庭网关在家庭中起到总控、协调所有设备的作用,并对用户提供统一、方便的使用界面。图9-1是智能网关的控制示意图。

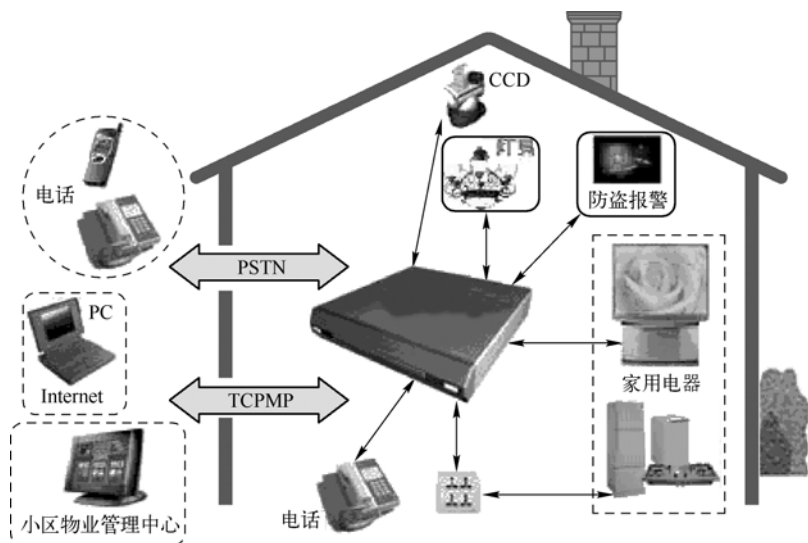


图 9-1 智能网关控制示意图

系统通过一台控制主机(智能网关)或相关控制设备(遥控器)实现对室内外整体照明、用电器、安全防范设备的集中控制,统一管理。它包括设备的本地开关控制、调光软起、红外控制、定时控制、电话远程控制、无线遥控控制、小区信息集中发布、远程网络监控、集中及场景模式组合等多种控制方式。

9.2 功能需求

要求开发的产品具备如下功能。

1. 网关功能

(1) 作为家庭网络与外部网络(Internet、PSTN、GPRS)的连接设备。实现网络之

间信息的互通，做到通过计算机、PDA、手机、固定电话等通信终端对家庭自动化设备的控制。

(2) 可以在各种网络上传输，并且可以通过 ADSL 独立拨号上网；提供动态域名解析功能。

(3) 浏览：B/S 浏览模式。

(4) 可以通过网络对家庭智能化网关的参数进行远程配置。

(5) 可以通过电话（拨打电话或发送短信）对家庭智能化网关进行简单配置（如远程撤、布防，打开和关闭预设的一些控制状态）。

(6) 家庭内部控制网络可以采用射频+红外或总线两种方式。

(7) 联网：可以任意指定一组网关组建网上虚拟社区，并和管理中心交换信息（如报警信息的集中管理）。

(8) 升级：可以通过局域网或广域网对指定网关进行升级。

2. 音视频监控功能

(1) 4 路音视频（NTSC/PAL）的输入（其中 1 路可接数字电视机顶盒的输入）。

(2) 本地视频组合显示：4 路画面全屏显示；母子画面显示功能；子画面可以在屏幕上随意移动；子画面可以改变大小；母子画面可以随意来回切换；四分画面显示。

(3) 1 路输出（采用 TV 做监视器）。

(4) 达到 D1 显示效果，分辨格式可调：单路 D1，屏蔽了其他 3 路的显示；两路 2CIF，屏蔽了其他两路的显示；三、四路 CIF，屏蔽了其他路的显示。

(5) MPEG4 算法，压缩率高，单路视频全实时占用带宽小于 100kb。

(6) 动静态侦测：在一个视频源上，定义多个动静态侦测区域，灵敏度可调。

(7) 本地设置及控制：采用红外遥控控制。

(8) 视频：支持 D1、H-D1、CIF、QCIF 输出格式，能够实现 N/P 制式自动切换。

(9) 音频：立体声，网络对讲。

(10) 串口：可以外挂各种设备，拥有透明通道。例如，支持远程云镜控制。

(11) 级联和扩展性：当输入多于 4 路时，可以扩充。

(12) 报警：可以连接各种信号（开关量）采集设备（16 路有线、8 路无线），报警信息可以传到网络另一端用户指定的终端，如信箱（两个以上信箱可设）、手机、固定电话（两个以上手机号码、三个以上固定电话可设）、蜂鸣器（可以设定开关）等，联动监控现场图片 E-mail 指定信箱；发送监控现场图片（彩信）到指定的手机，也可以发送到虚拟网络社区中的管理中心。注意：报警模块采用外挂方式设计，主机（网关）配置专用报警模块 I/O 接口。

(13) 撤布防遥控器（RF）可进行布防、撤防、紧急报警等操作。布防按键有 3 个：离家模式、在家模式、休息模式。

(14) 报警设置：可以通过网关专用遥控器设置。[应考虑如下功能设计：如何进入系统设置状态；设置和修改密码；修改外出延时时间；修改进入延时时间；修改防区类型；撤布防遥控器学习方法（配无线模块使用）；无线探测器的学习方法（配无线模块使用）；删除防区操作，对于不用的防区可以删除；退出系统设置状态；复位操作，把模块恢复到初始状

态；如何实现离家模式、在家模式和休息模式下的防区选择]。

(15) OSD：支持时间显示、中文信息显示、图标显示（在监视的时候可以设定为打开或者关闭）。

(16) 存储：内置闪存，用于存储报警图片（最少 64 张），或配置 SD 卡插槽或者其他的 USB 存储设备；支持远程存储功能，可以设定时间段录像，设定报警和运动侦测触发录像。

(17) 日志：用户可以直观查到报警日志、移动侦测日志、视频丢失日志和系统日志。

(18) 权限管理：设定管理员和用户两级权限。

3. 家电自动控制功能

(1) 智能开关、智能插座通过智能网关进行控制，网关可发射无线射频信号。

(2) 无线/红外转发器可将网关无线信号转发成红外信号，实现对红外家电进行控制。

(3) 组合控制：把任意几种家电设备的单独功能组合起来作为一个功能，实现对多个设备的联动控制。

(4) 条件控制：家庭智能化网关可以外接各种信号源，根据设定条件，控制一种或几种家电设备的动作。例如，智能网关具有室内温度检测、显示功能，室温检测与空调实现条件控制功能。

(5) 电话远程控制：采用密码、用户名验证系统；智能网关嵌入电话语音模块，通过用户电话线路实现电话报警功能；通过网关电话语音模块可进行远程布防等安全操作；通过网关电话语音模块对所有家电、照明系统进行远程电话控制。

(6) Internet 远程控制：通过 Internet 对家居内的家电、灯光、插座、安防探测器进行远程网络控制；通过 Internet 实现对家居内的安防系统进行远程撤布防、消警、报警查询等功能；实现远程控制可视化操作。

4. 控制界面功能

(1) 家电控制界面设计人性化，操作简便。

(2) 方便的云台和摄像头控制，支持预置位管理和方便的快捷键控制。

(3) 支持语音双向对讲。

(4) 支持 3 种录像方式：定时、报警、运动侦测。

(5) 时间调度表功能，按星期安排网关的工作模式。

(6) 录像回放。

(7) 回放程序支持正常，快进和跳进三种播放模式，并且支持回放过程中的抓图功能

(8) 选定要控制的设备的同时监视图像自动切换到对应区域的摄像机画面下。

9.3 家庭网关设计

家庭网关的设计包括硬件和软件两部分。首先是硬件部分的设计，硬件要求能对所有的功能提供支持，一般是在评估板上进行。本产品由于有视频功能，因此需要专用的视频处理部件。有多种选择，前几年由于 ARM 的速度不是很高，采用专用视频芯片和 ARM 结合的方式，专用芯片的优点是对 MPEG4、H.264 等直接提供硬件的支持。目前由于技术的发展，ARM cortex A8/A9 处理器已经达到 GHz 的速度。如三星的 A8 处理器 S5PV210 已经可

以直接处理视频，但总体来说毕竟还是单核处理器。而 A9 则具有多核，处理视频比较方便。当然，这些主频很高，进行 PCB 布线时要注意高速布线的问题。高速布线其实就是电路理论中的传输线理论，直观点讲就是高频信号在导线上传输，不同的距离电压不同。因此必须控制传输线参数，否则信号正常传输时 CPU 跑不起来。本书针对的是起步较低的人员，所以不准备通过 Cortex A8/A9 处理器讲解，而以 ARM 9 的 S3c2410 为例，去除视频芯片，专注于介绍在 Linux 下的普通嵌入式应用开发，等这些知识掌握以后，再学习视频芯片开发就比较容易了。

9.3.1 网络通信设计

网络通信设计是家庭网关设计中的主要部分，采用 10MB 以太网控制芯片，以太网前端接路由器，路由器完成 NAT 转换。在网关内部运行 Web 服务器进程，所有的控制参数都存放在嵌入式数据库中，用户通过浏览器对 Web 页面进行设置的数据都将存入数据库。

图 9-2 是家庭网关的网络结构图。家庭网关控制器可以通过局域网、广域网来远程访问和控制。

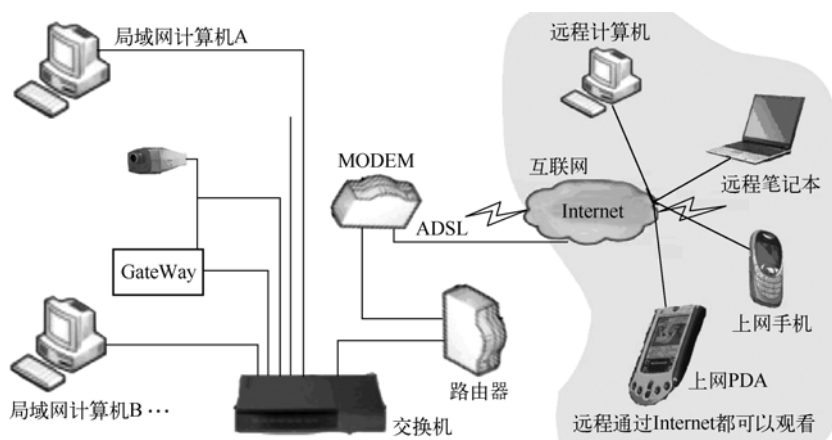


图 9-2 家庭网关的网络结构图

网关需要通过远程浏览器进行设置，如果通过 ADSL 上网，因为地址是动态的，需要外界知道目前动态分配的 IP 地址，为此，采用如图 9-3 和图 9-4 所示的方式解决此问题。

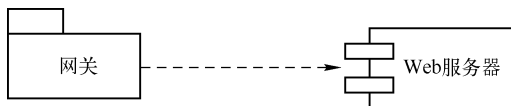


图 9-3 动态分配 IP 地址

根据上述方式，用户可以通过在某个固定域名的网站上输入用户名和密码直接转到自己家里的网关，从而实现远程管理。

网关接入网络的类型目前主要有两种，一种是永久在线的，另一种是需要拨号的。对于类似有线通的用户，不存在拨号问题。而对于 ADSL，在网关软件中包含了 ADSL 拨号功

能，网关上电后就能自动拨号。软件包含了网络类型的识别，可根据不同的类型予以处理。对于小区局域网或企业内部网络，则必须采用隧道技术，否则可能无法在外部访问。

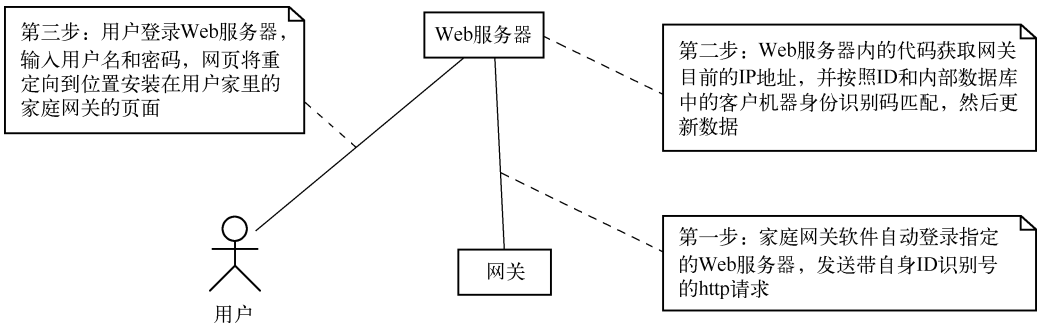


图 9-4 IP 解析

9.3.2 网关软件架构

网关的管理采用 B/S 方式实现。在网关内部有一个 Web 服务器，所有的业务逻辑全部用 CGI 实现。图 9-5 是网关软件的组件结构。

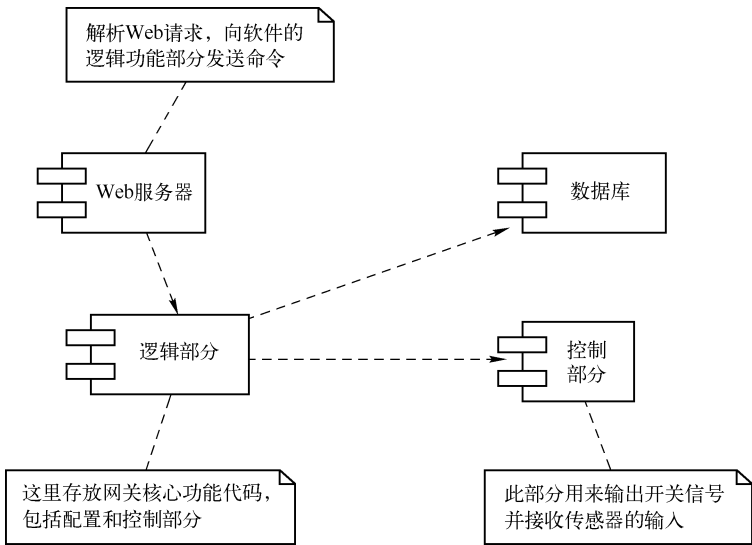


图 9-5 网关软件的组件结构

从应用层面的角度考虑，网关主要包括 Web 服务器组件、数据库、业务逻辑和控制逻辑。Web 服务器用来解析 http 请求和命令，并将数据提交给业务逻辑组件，将业务逻辑返回的数据以 XML 的形式提交给用户。业务逻辑组件完成主要的业务功能，将数据保存到数据库或者从数据库检索出数据，根据用户的安排将执行指令发送给控制组件。控制组件负责驱动相应的设备并置相应的状态。数据库存放各种配置、检测数据，以及日志记录。

报警实现：报警信息包含文字和图片两种。图片以 JPG 方式保存并发送，发送通过

SMTP 协议发邮件或者连接彩信发送平台。

9.3.3 关于视频硬件设计

视频的硬件结构如图 9-6 所示。

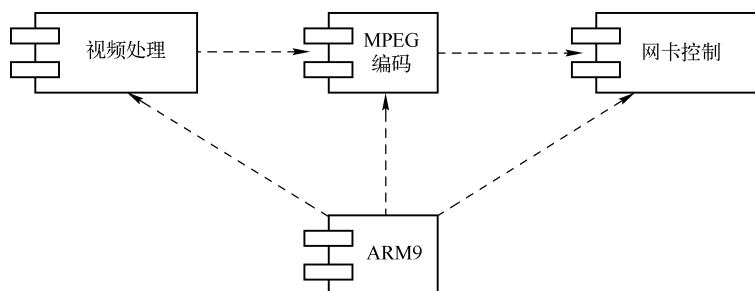


图 9-6 视频的硬件结构

1. 发送端

视频信号输入视频处理器芯片，其输出的 A/D 经过 MPEG4 编码器编码，MPEG4 编码输出的数据经过 ARM 9 封装，送入网络控制芯片。其中，ARM 9 和 maccontroller 采用 DMA 传输，以保证数据传输的实时性。

2. 接收端

接收端应用程序对 MPEG4 进行解码，并且播放。在应用程序中采用 Windows Media Player 组件。

9.3.4 系统整体的硬件设计

家庭网关包括的硬件总体结构如图 9-7 所示。

对应到具体的设备，家庭网关包含的硬件如下：

- 485 接口 1 个；
- 232 接口 1 个；
- 红外接口 1 个；
- 无线接口 1 个；
- SD 卡接口；
- 网络；
- 声卡；
- 温度控制接口 1 个；
- GSM 接口；
- PSTN 接口；
- 视频采集及压缩硬件；
- ARM 处理器；
- 外围 I/O 采集转接 485 电路板；
- USB 接口。

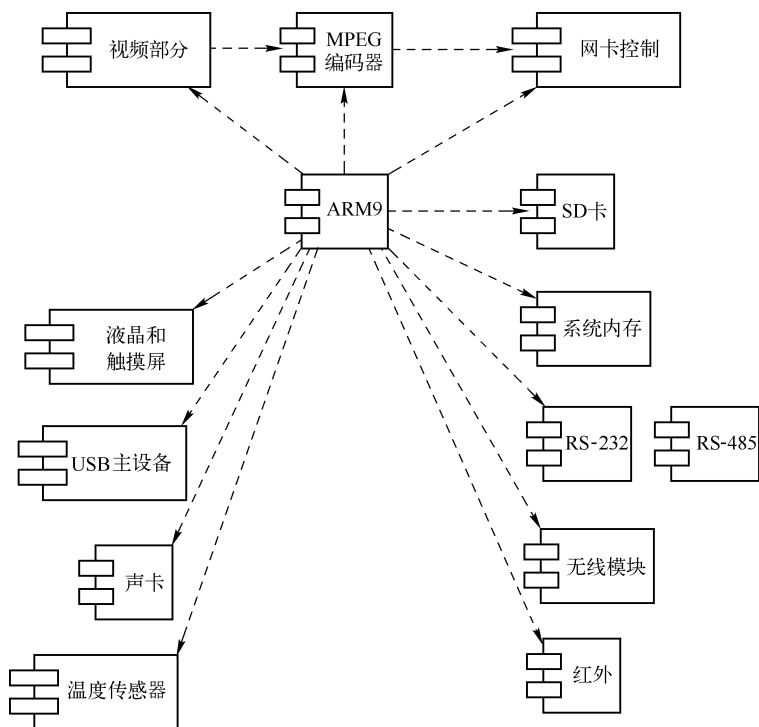


图 9-7 家庭网关的硬件总体结构

9.3.5 家庭网关系统包含的软件

家庭网关的软件系统主要由三部分构成，分别是：

- 网关内部软件；
- 客户端软件；
- 网站软件。

1. 网站软件

网站软件为 ASP 代码，具体包括：

- IP 地址获取；
- 用户登录重定向；
- IP 动态数据库更新；
- 软件自动更新区；
- 公共设置参数界面；
- 邮件系统。

2. 客户端软件

客户端软件包括：

- 房间选择；
- 日期提示；
- 留言信箱；

- 灯光、窗帘、场景；
- 个性化设置；
- 温度设置；
- 时间自动功能；
- 歌曲播放；
- 视频监控；
- 4 路音视频的显示，视频组合显示 4 路画面全屏显示、母子画面显示、子画面可以在屏幕上随意移动、子画面可以改变大小、母子画面可以随意来回切换、4 分割画面显示；
- 家电控制界面设计人性化，操作简便；
- 方便的云台和摄像头控制，支持预置位管理和方便的快捷键控制；
- 支持语音双向对讲；
- 支持 3 种录像方式——定时、报警、运动侦测；
- 时间调度表功能，按星期安排网关的工作模式；
- 录像回放；
- 回放程序支持正常，快进和跳进三种播放模式，并且支持回放过程中抓图功能；
- 选定所要控制设备的同时监视图像自动切换到对应区域的摄像机画面下。
- 网络安防。

3. 网关软件

网关软件包括以下几个部分。

1) 控制模块

控制模块包括输入/输出 (I/O)，具体有红外驱动、无线接收驱动、串口驱动、16 路开关量输入、8 路无线、远程云台驱动、智能开关驱动。

2) 代理模块

代理模块包括电话语音模块驱动、短信息驱动、彩信驱动、Web 服务代理、ADSL 自动拨号程序。

3) 业务模块

- MPEG4 算法：压缩率高，单路视频全实时占用带宽少于 100KB。
- 动静态侦测：在一个视频源上，定义多个动静态侦测区域，灵敏度可调。
- 本地设置及控制：采用红外遥控控制。
- 视频：支持 D1、H-D1、CIF、QCIF 输出格式，能够完成 N/P 制式自动切换。达到 D1 显示效果，分辨格式可调：单路 D1，屏蔽了其他 3 路的显示；两路 2CIF，屏蔽了其他两路的显示；三、四路 CIF，屏蔽了其他路的显示。
- 音频：立体声，网络对讲。
- 报警：可以连接各种信号（开关量）采集设备（16 路有线、8 路无线），报警信息可以传到网络另一端用户指定的终端，如信箱（两个以上信箱可设）、手机、固定电话（两个以上手机号码、三个以上固定电话可设）、蜂鸣器（可以设定开关）等，联动监控现场图片 E-mail 指定信箱；发送监控现场图片（彩信）到指定的手机，也可以

发送到虚拟网络社区中的管理中心。注意：报警模块采用外挂方式设计，主机（网关）配置专用报警模块 I/O 接口。

- 撤布防遥控器（RF）进行布防、撤防、紧急报警等操作。布防按键有 3 个：离家模式、在家模式、休息模式。
- 报警设置：可以通过网关专用遥控器设置。
- OSD：支持时间显示、中文信息显示、图标显示（在监视的时候可以设定为打开或者关闭）。
- 存储：内置闪存，用于存储报警图片（最少 64 张），或配置 SD 卡插槽或者其他的 USB 存储设备；支持远程存储功能，可以设定时间段录像，设定报警和运动侦测触发录像。
- 日志：用户可以直观查到报警日志、移动侦测日志、视频丢失日志和系统日志。
- 权限管理：设定管理员和用户两级权限。
- 组合控制：把任意几种家电设备的单独功能组合起来作为一个功能，实现对多个设备的联动控制。
- 条件控制：家庭智能化网关可以外接各种信号源，根据设定条件，控制一种或几种家电设备的动作。例如，智能网关具有室内温度检测、显示功能，室温检测与空调实现条件控制功能。
- 电话远程控制：采用密码、用户名验证系统；智能网关嵌入电话语音模块，通过用户电话线路实现电话报警功能；通过网关电话语音模块可进行远程布防等安全操作；通过网关电话语音模块对所有家电、照明系统进行远程电话控制。
- Internet 远程控制：通过 Internet 对家居内的家电、灯光、插座、安防探测器进行远程网络控制；通过 Internet 实现对家居内的安防系统进行远程撤布防、消警、报警查询等功能；实现远程控制可视化操作。

9.4 硬件平台设计

硬件首先在开发板上测试，等开发板测试通过后，再进一步布电路板，使其成为真正的产品。网关硬件平台使用三星 S2C2410 微处理器的开发板，该开发板由一块核心板和一块母板构成。具体包括：

- SAMSUNG S3C2410 微处理器；
- CS8900A 以太网控制芯片；
- Hynix HY57V561620CT-H SDRAM 16M*8bit*2；
- SAMSUNG K9F1208U0B 64M*8bit NAND Flash；
- JTAG 端口；
- RJ-45 以太网口；
- RS-232 串口；
- RS-485 串口；
- 电源接口。

下面介绍主要的硬件芯片。

1. S3C2410 微处理器

S3C2410 处理器是三星公司基于 ARM 公司的 ARM920T 处理器核，采用 0.18 μ m 制造工艺的 32 位微控制器。该处理器拥有独立的 16KB 指令 Cache 和 16KB 数据 Cache，MMU，支持 TFT 的 LCD 控制器，NAND 闪存控制器，3 路 UART，4 路 DMA，4 路带 PWM 的 Timer，I/O 口，RTC，8 路 10 位 ADC，Touch Screen 接口，IIC-BUS 接口，IIS-BUS 接口，两个 USB 主机，一个 USB 设备，SD 主机和 MMC 接口，两路 SPI。S3C2410 处理器最高可运行在 203MHz。

(1) S3C2410 芯片集成了大量的功能单元，包括：

- 内部 1.8V，存储器 3.3V，外部 I/O 3.3V，16KB 数据 Cache，16KB 指令 Cache，MMU。
- 内置外部存储器控制器（SDRAM 控制和芯片选择逻辑）。
- LCD 控制器，一个 LCD 专业 DMA。
- 4 个带外部请求线的 DMA。
- 3 个通用异步串行端口（IrDA1.0、16-Byte Tx FIFO and 16-Byte Rx FIFO），2 通道 SPI。
- 一个多主 I²C 总线，一个 I²S 总线控制器。
- SD 主接口版本 1.0 和多媒体卡协议版本 2.11 兼容。
- 两个 USB HOST，一个 USB DEVICE（VER1.1）。
- 4 个 PWM 定时器和一个内部定时器。
- 看门狗定时器。
- 117 个通用 I/O。
- 56 个中断源。
- 24 个外部中断。
- 电源控制模式——标准、慢速、休眠、掉电。
- 8 通道 10 位 ADC 和触摸屏接口。
- 带日历功能的实时时钟。
- 芯片内置 PLL。
- 设计用于手持设备和通用嵌入式系统。
- 16/32 位 RISC 体系结构，使用 ARM920T CPU 核的强大指令集。
- 带 MMU 的先进的体系结构支持 WinCE、EPOC32、Linux。
- 指令缓存（Cache）、数据缓存、写缓存和物理地址 TAG RAM，减小了对主存储器带宽和性能的影响。
- ARM920T CPU 核支持 ARM 调试的体系结构。
- 内部先进的位控制器总线（AMBA）（AMBA2.0，AHB/APB）。

(2) 系统管理，包括：

- 小端/大端存储支持。
- 地址空间：每个 BANK 为 128MB（全部为 1GB）。

- 每个 BANK 可编程为 8/16/32 位数据总线。
- BANK0~BANK6 为固定起始地址。
- BANK7 可编程 BANK 的起始地址和大小。
- 一共有 8 个存储器 BANK。
- 前 6 个存储器 BANK 用于 ROM、SRAM 和其他。
- 两个存储器 BANK 用于 ROM、SRAM 和 SDRAM（同步随机存储器）。
- 支持等待信号用于扩展总线周期。
- 支持 SDRAM 掉电模式下的自刷新。
- 支持不同类型的 ROM 用于启动（NOR/NAND Flash、EEPROM 和其他）。

(3) 芯片封装：

- 272-FBGA 封装。

2. CS8900A 以太网控制芯片

CS8900A 芯片是 Cirrus Logic 公司生产的一种局域网处理芯片，在嵌入式领域中的使用非常广泛。它的封装是 100-pin TQFP，内部集成了片内 RAM、10BASE-T 收发滤波器，并且提供 8 位和 16 位两种接口。

3. RJ-45 以太网口

本网关的以太网口主要用于通过以太网网络访问网关的 Web 页面，同时也可用于通过超级终端等软件以 TCP/IP 的连接方式来连接开发板，这样能进行更稳定、更快速的文件传输和调试工作。

4. RS-232 和 RS-485 串口

在本网关的开发中，RS-232 接口主要用于连接 PC，以便通过超级终端等软件来连接网关主机进行开发工作。而 RS-485 接口则用于连接转发器，网关发送的任何指令都通过此端口发往转发器，并由转发器完成指令的发送工作。

9.5 嵌入式 Web 开发概述

网关基于 Mizi Linux 嵌入式操作系统平台，使用 Boa 作为 Web 服务器，使用 SQLite 轻型数据库作为后台管理数据库。后台 CGI 脚本使用 C/C++ 编写，前台 Web 页面则使用 HTML、JavaScript 和 CSS 编写，利用 Ajax 技术实现前台与后台之间的数据通信。

9.5.1 Mizi Linux

Mizi 公司开发的 Mizi Linux 使用 Linux 2.4 内核，支持 S3C2410 微处理器，并提供了完整的适用于 S3C2410 的 Mizi Linux SDK，极大地方便了开发人员基于 S3C2410 和 Mizi Linux 平台的嵌入式应用程序开发。

9.5.2 Boa 小型 Web 服务器

Boa 是一个单任务的 HTTP Web 服务器，支持认证、CGI 脚本等，功能比较全，性能比较高。它和传统 Web 服务器的主要区别是：它是单进程的，在接到新的 HTTP 请求时并不

FORK 出一个新的进程来响应，不能同时处理多个到来的连接，而且也不能将自己复制多个副本来处理多连接。它在内部处理所有正在进行的 **HTTP** 连接请求，只对单独的 **CGI** 程序、自动文件的产生、自动文件的解压等请求 **FORK** 出新进程。它的优点是代码简单、速度快、适合于嵌入式应用。

值得一提的是，**Boa** 是一款完全开放源码的免费软件，用户可以随时从官方网站上获取 **Boa** 最新版本并编译运行于不同的操作系统平台。

9.5.3 SQLite 轻型数据库

SQLite 是一个轻型的嵌入式数据库，同样它也是一款源代码级的免费软件。**SQLite** 不同于其他大部分的 **SQL** 数据库引擎，因为它的首要设计目标就是简单化：易于管理、易于使用、易于嵌入其他大型程序、易于维护和配置。

小巧、快速、稳定是 **SQLite** 的优势。出色的稳定性源于 **SQLite** 力争做到简单化，越简单就越不容易出错。也正是因为 **SQLite** 的这些优势，**SQLite** 非常适合运行于资源有限的嵌入式平台上。此外，因为 **SQLite** 数据库几乎不需要管理，因此对于那些无人值守运行或无人工技术支持的设备或服务，**SQLite** 是一个很好的选择。**SQLite** 能很好地适用于手机、PDA、机顶盒等嵌入式设备；作为一个嵌入式数据库，它也能很好地应用于客户端程序。

9.5.4 网关的软件平台构造

如图 9-8 所示，基于 **S3C2410** 和 **Mizi Linux** 平台之上，**Boa** 作为 **Web** 服务器担负着与客户端通信的重要工作。当接收到客户端浏览器发送的 **HTTP** 请求后，**Boa** 发送相应的 **HTML**、**JavaScript** 和 **CSS** 文件到客户端浏览器。用户在浏览器网页上进行相应互动操作后，**JavaScript** 使用 **Ajax** 技术与 **Boa** 进行通信，**Boa** 会调用相应的 **CGI** 脚本响应用户的请求，并发回响应的结果。同时，**CGI** 脚本可以直接访问、操作 **SQLite** 后台数据库或者网关 **I/O** 端口，以获取相关数据或发送相应的控制指令。

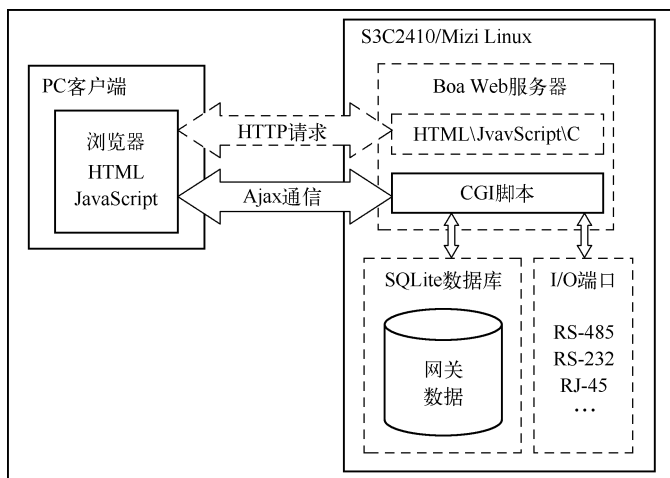


图 9-8 软件平台构造图

9.6 软件环境搭建步骤

9.6.1 烧写 Mizi Linux

由于 Linux 移植已超出本章讨论范围, 故在此只给出在 S3C2410 开发板上移植 Linux 系统的大致步骤, 如下所示。

(1) 通过 JTAG 接口烧写 VIVI 启动代码。

(2) 在开发板上运行 `load flash kernel x` 命令, 并通过串口 `xmodem` 方式传送 Linux 内核文件 (`zImage`), 完成 Linux 内核烧写。

(3) 在开发板上运行 `load flash root x` 命令, 并通过串口 `xmodem` 方式传送 Linux 根文件系统文件 (`root.cramfs`), 完成根文件系统烧写。

9.6.2 搭建交叉编译环境

要进行嵌入式设备应用程序开发, 在 x86 Linux 上安装交叉编译环境是首要条件。

如前文所述, MIZI 公司提供了完整的适用于 S3C2410 的 Mizi Linux SDK。在用于开发的 x86 Linux 系统上解压缩 SDK 压缩包后, 依次安装所有的 RPM 包就可以完成交叉编译环境的安装。安装完成后, 所有的交叉编译工具都会存在于 x86 Linux 的 `/opt/host/armv4l/bin/` 目录之下, 如 `armv4l-unknown-linux-gcc`、`armv4l-unknown-linux-g++` 和 `armv4l-unknown-linux-strip` 等。

为了使系统每次启动后用户都能直接调用这些交叉编译工具, 需要把 `/opt/host/armv4l/bin/` 添加到系统路径 `PATH` 内, 具体操作步骤如下 (以 Ubuntu Linux 8.04 为例)。

(1) 在终端控制台中输入命令 `sudo vi /etc/bash.bashrc`。

(2) 在后面加入如下代码:

```
if [ -d /usr/local/arm ] ; then
    PATH=/usr/local/arm/bin:${PATH}
fi
```

(3) 在终端控制台中输入命令 `# source /etc/profile`, 使新的环境变量生效。

至此, 就完成了交叉编译环境的安装。读者可以自行编写一个简单的 Hello World 程序, 尝试使用交叉编译器进行编译, 并将生成的二进制文件上传至开发板进行测试。

9.6.3 Boa 移植

Boa 是一款源代码开放的免费 Web 服务器, 其移植步骤如下。

(1) 从 Boa 的官方网站 (<http://www.boa.org>) 上获取最新版的软件源代码, 目前最新版为 0.94.13。

(2) 解压下载的源代码压缩包。

(3) 进入解压目录的 `src/` 目录下, 输入 `./configure` 命令, 生成 Makefile 文件。

(4) 修改 src/compat.h 文件, 将

```
#define TIMEZONE_OFFSET(foo) foo##->tm_gmtoff
```

修改为

```
#define TIMEZONE_OFFSET(foo) (foo)->tm_gmtoff
```

(5) 运行 make 命令, 即可生成运行于 x86 Linux 平台的 Boa 二进制文件。

(6) 修改 boa.c 文件, 将

```
if (setuid(0) != -1) {
    DIE("icky Linux kernel bug!");
}
```

三行语句全部注释掉。

(7) 修改 Makefile 文件, 将

```
CC = gcc
CPP = gcc -E
```

改为

```
CC = armv4l-unknown-linux-gcc
CPP = armv4l-unknown-linux-gcc -E
```

(8) 运行 make clean, 再运行 make 命令, 即可生成运行于 ARM Linux 平台的 Boa 二进制文件。

至此, 能够运行于 S3C2410/Mizi Linux 平台的 Boa 已经成功生成, 可将该 Boa 二进制文件上传至开发板。但是此时还无法启动 Boa, 因为还未对 Boa 进行配置, 需要对 boa.conf 文件进行相应的配置, 方法见步骤 (9)。

(9) 回到 x86 Linux 上的 Boa 解压目录下, 打开 boa.conf 文件, 修改如下配置:

- Port——设置提供 Web 服务的端口号。这里使用默认端口号 80。
- User——设置以什么用户身份访问 Web。因为需要访问并运行 CGI 脚本, 为了防止权限不足的问题, 这里使用 root 作为 Web 访问身份。
- Group——设置访问用户所属用户组。因为嵌入式设备上无 nogroup 组, 故这里设置成 0。
- ErrorLog——错误日志存放路径。因为在开发板上 Boa 放在/mnt/下, 故这里设置成 /mnt/error_log。
- AccessLog——访问日志存放路径。同 ErrorLog, 设置成/mnt/access_log。
- DocumentRoot——Web 根文件夹, 即存放网站首页 HTML 文件的文件夹。这里设置成/mnt。
- DirectoryIndex——默认首页 HTML 文件文件名。这里设置成 index.html。
- DirectoryMaker——创建文件夹列表的文件夹路径。这里设置成/mnt。
- MimeTypes——设置 mime.types 文件路径。这里设置为/mnt/mime.types。

- CGIPath——设置 CGI 脚本的存放路径。这里设置为 `/mnt/cgi-bin`。
- ScriptAlias——设置 CGI 脚本文件夹别名。这里设置为 `ScriptAlias/cgi-bin//mnt/cgi-bin/`。

(10) 将修改好的 `boa.conf` 和 `mime.types`（一般位于 `/etc/` 下）文件传到开发板 Boa 所处的同文件夹（即 `/mnt/`）下。

(11) 逐一建立在 `boa.conf` 文件中设置的各个文件夹（包括所有涉及的子文件夹）。

(12) 运行 `./boa -c` 命令，正常情况下即可启动 Boa。可以使用 `ps -A` 命令确认 Boa 进程是否已启动。

(13) 用户可以建立一个简单的 `index.html` 文件并传输到开发板 `/mnt/` 文件夹下，然后在客户端使用浏览器[输入地址 `http://网关 IP 地址:端口号`（默认 80 端口可不输）]测试 Boa 是否能正常工作。

9.6.4 SQLite 移植

SQLite 是一款源代码开放的免费轻量级数据库软件，其移植步骤如下（假设已按照 2.3.2 节的方法成功搭建了交叉编译环境）。

(1) 从 SQLite 的官方网站 (<http://www.sqlite.org>) 上获取最新版的软件源代码，目前最新版为 3.6.5，本文使用的版本为 3.5.9，两者的编译方法相同。

(2) 解压下载的源代码压缩包，假设解压在文件系统根目录下 (`/`)。

(3) 在文件系统根目录下建立一个文件夹，以存放生成的文件 `mkdir /sqlite-arm`。

(4) 进入解压目录，运行命令 `./configure --host=armv4l-unknown-linux --prefix=/sqlite-arm --disable-tcl`，即可在解压目录下生成适用于 ARM Linux 平台的 Makefile 文件。

(5) 运行 `make` 命令，系统就会利用交叉编译器来编译 SQLite。

(6) 运行 `make install` 命令。

如果以上步骤都顺利完成，则在 `/sqlite-arm` 文件夹下会生成以下 3 个文件夹。

- `bin/`，包含运行于 ARM Linux 平台的 SQLite3 二进制文件。
- `include/`，包含 `sqlite3.h` 和 `sqlite3ext.h` 两个头文件。
- `lib/`，包含 SQLite3 适用于 ARM Linux 平台的几个链接库文件，如动态链接库文件 `libsqlite3.so.0.8.6` 和静态链接库文件 `libsqlite3.a` 等。

值得指出的是，在交叉编译有关 SQLite 的应用程序时，如果采用普通的动态链接方式编译，则除了编译好的二进制文件外，位于该 `lib/` 目录下的几个库文件也需要上传至开发板，该程序才能正常运行，否则会因为找不到相应的 SQLite 链接库文件而无法运行。所以为了避免麻烦，在编译 ARM 版本的 SQLite 应用程序时一般采用静态链接的方式进行，这样生成的二进制文件虽然比较大，但却不需要提供额外的 SQLite 链接库文件。

此外，因为编译生成的 SQLite 二进制文件和库文件 `libsqlite3.so.0.8.6` 中包含有调试信息，所以文件体积比较大，可以使用 `strip` 工具剥离调试信息，方法见步骤 (7)。

(7) 进入 `/sqlite-arm`，运行 `armv4l-unknown-linux-strip bin/sqlite3` 命令和 `armv4l-unknown-linux-strip lib/libsqlite3.so.0.8.6` 命令来剥离调试信息，生成体积较小的二进制文件。

至此，就完成了 ARM Linux 平台的 SQLite 的编译。在实际的嵌入式开发过程中，一般

不直接在开发板上使用 SQLite3 程序来操作和管理数据库。为了方便操作，通常都在 x86 Linux 上建立数据库并添加相应的数据表和数据，并将生成的数据库文件上传至开发板。所以有必要再编译一个能够运行于 x86 Linux 平台的 SQLite 程序，以方便在 x86 Linux 上操作数据库文件。

要编译能够运行于 x86 Linux 平台的 SQLite，方法更为简单，依次按照上述的 ARM 版本编译方法执行到第（6）步即可，其中需要注意：跳过第（3）步；在第（4）步中只需运行 `./configure` 命令，无须其他开关参数。这样就完成了 x86 版 SQLite 的编译，而且在执行 `make install` 命令后，x86 版的 SQLite3 二进制文件被存放在 `/usr/local/bin/` 目录下，而 x86 版的链接库文件则存放在 `/usr/local/lib/` 目录下，由于这两个路径都默认存在于 `PATH` 环境变量中，所以用户可以直接调用 SQLite3 命令或相应的 `sqlite` 链接库文件，相当方便。

9.6.5 SQLite 使用方法与常用命令

完成了 x86 Linux 版 SQLite 的编译后，有必要简单介绍一下 SQLite 的使用方法和常用命令，以方便日常数据库的创建、修改、调试和维护。

可以使用 `sqlite3 test.db` 命令进入 SQLite3 的命令行操作界面。如果不存在 `test.db` 文件则创建之，反之则打开。成功进入 SQLite3 命令行界面后会显示如下信息：

```
SQLite version 3.5.9
Enter ".help" for instructions
sqlite>
```

在 `sqlite>` 提示符后可以输入常规的 SQL 语句，并以分号（`;`）结尾，回车即可执行相应的 SQL 语句。如 `sqlite> select*from test;` 表示查询 `test` 表中的所有数据记录。

除了常规 SQL 语句外，SQLite 还内置了一些指令方便对数据库进行管理，如下所示：

<code>.databases</code>	列出数据库文件名
<code>.tables ?PATTERN?</code>	列出与 <code>?PATTERN?</code> 匹配的表名
<code>.import FILE TABLE</code>	将文件中的数据导入文件中
<code>.dump ?TABLE?</code>	生成形成数据库表的 SQL 脚本
<code>.output FILENAME</code>	将输出导入指定的文件中
<code>.output stdout</code>	将输出打印到屏幕
<code>.mode MODE ?TABLE?</code>	设置数据输出模式（ <code>csv</code> ， <code>html</code> ， <code>tcl</code> ...）
<code>.nullvalue STRING</code>	用指定的串代替输出的 NULL 串
<code>.read FILENAME</code>	执行指定文件中的 SQL 语句
<code>.schema ?TABLE?</code>	打印创建数据库表的 SQL 语句
<code>.separator STRING</code>	用指定的字符串代替字段分隔符
<code>.show</code>	打印所有 SQLite 环境变量的设置
<code>.help</code>	显示所有的可用命令和帮助信息
<code>.quit</code>	退出命令行接口

注意：以上 SQLite 内置指令前均有一个“`.`”，且指令后也无须加分号“`;`”，只有执行


```
CREATE TABLE roomset (
  id integer primary key unique check (id>=0 and id<16), ——编号, 0~15, 共 16 个
  roomid tinyint not null default 0); ——房间编号, 来自 roominfo, 0 表示未设置房间
```

示例数据:

```
INSERT INTO "roomset" VALUES(0,1); ——房间 0 设置为“客厅”
INSERT INTO "roomset" VALUES(1,2); ——房间 1 设置为“饭厅”
```

3. appcmdlist 表

该表用于存放每一种家电设备的各条指令名称字符串。例如, 用于 DVD 控制的指令有“开/关”、“出盒”、“播放”、“暂停”、“停止”等。

```
CREATE TABLE appcmdlist (
  apptype tinyint check(apptype>=0 and apptype<5), ——家电类型, 0~5
  cmdindex tinyint not null, ——该指令在当前家电指令列表中的索引号
  cmdname varchar(20) not null, ——指令名称, 如“开/关”、“播放”等
  primary key(apptype,cmdindex));
```

示例数据:

```
INSERT INTO "appcmdlist" VALUES(0,0,'开/关');
INSERT INTO "appcmdlist" VALUES(0,1,'静音');
```

4. appname 表

该表用于存放用户已添加的家电设备的信息, 以及每个家电对应的信息转发器编号。

```
CREATE TABLE appname (
  apptype tinyint check(apptype>=0 and apptype<5), ——家电类型
  roomid tinyint, ——房间编号
  appid tinyint check(appid>=0 and appid<256), ——家电编号, 每个房间可有 256 个家电
  appname varchar(20) not null, ——家电名称
  routeid smallint check(routeid>=0 and routeid<16) not null, ——转发器编号
  primary key(roomid,appid));
```

示例数据:

```
INSERT INTO "appname" VALUES(1,1,0,'海尔空调',14); ——在客厅添加“海尔空调”
INSERT INTO "appname" VALUES(4,2,0,'北窗帘',12); ——在饭厅添加“北窗帘”
```

5. appctrlinfo 表

该表用于存放各个家电设备已经学习的遥控指令信息。

```
CREATE TABLE appctrlinfo(
  appcmd int, ——已学习的遥控指令, 4 字节长度
  cmdindex tinyint, ——该指令在当前家电指令列表中的索引号
  appid tinyint check(appid>=0 and appid<256), ——家电编号
  roomid tinyint, ——房间编号
```

```
apptype tinyint check(apptype>=0 and apptype<5),——家电类型
primary key(roomid,appid,cmdindex));
```

示例数据:

```
INSERT INTO "appctrlinfo" VALUES(1667457891,0,0,1,1); /*客厅中“海尔空调”的“开/关”
遥控指令信息*/
INSERT INTO "appctrlinfo" VALUES(1734702945,0,0,2,4); /*饭厅中“北窗帘”的“开”遥控
指令信息*/
```

6. switchinfo 表

该表用于存放目前已添加的灯光开关设备的信息。

```
CREATE TABLE switchinfo (
roomid tinyint not null,           ——房间编号
swid smallint check(swid>=0 and swid<4096), ——遥控器地址, 12 位
keyid tinyint check(keyid>=0 and keyid<16), ——遥控器按键编号, 0~15
routeid tinyint check(routeid>=0 and routeid<16), ——转发器编号
swname varchar(20),               ——灯光开关名称
primary key(roomid,swid,keyid,routeid,swname));
```

示例数据:

```
INSERT INTO "switchinfo" VALUES(1,4,7,14,'测试灯光'); ——客厅中添加“测试灯光”
INSERT INTO "switchinfo" VALUES(1,2,1,14,'test'); ——客厅中添加“test”
```

7. userpwd 表

该表用于存放用户设置的普通密码、超级密码和安防密码的 md5 码。

```
CREATE TABLE userpwd (
pwdtype smallint primary key, ——密码类型, 1—普通密码, 2—超级密码, 3—安防密码
pwd char(32)) ; ——密码的 md5 码
```

示例数据:

```
INSERT INTO "userpwd" VALUES(1,'e10adc3949ba59abbe56e057f20f883e'); ——普通密码
INSERT INTO "userpwd" VALUES(2,'e10adc3949ba59abbe56e057f20f883e'); ——超级密码
```

8. defvalues 表

该表用于存放网关的默认设置参数, 以便于用户恢复出厂设置。

```
CREATE TABLE defvalues (
key varchar[20] , ——设置名
value varchar[50]); ——设置默认值
```

示例数据:

```
INSERT INTO "defvalues" VALUES('userpwd','e10adc3949ba59abbe56e057f20f883e');
——默认普通密码的 md5 码
```

```
INSERT INTO "defvalues" VALUES('safepwd','e10adc3949ba59abbe56e057f20f883e');  
——默认超级密码的 md5 码
```

至此，目前网关所需使用的数据表已全部创建完毕。

9.7 CGI 程序设计与实现

本节主要就 CGI 脚本的设计和实现原理、方法等进行详细论述。

9.7.1 CGI 与客户端的通信机制

客户端浏览器通过 Ajax 技术来发送 CGI 请求，那么作为后台响应的 CGI 脚本是如何提取随同 CGI 请求一起发送的参数数据的呢？CGI 脚本又是如何将响应数据发送给客户端浏览器的呢？例如，客户端通过 Ajax 发出 CGI 请求“cgi-bin/getlamplist.cgi?roomid=1”，该请求的作用是通过 getlamplist.cgi 获取房间 1 中已添加的灯光设备列表。那么 getlamplist.cgi 是如何获取请求的参数 roomid=1 的呢？CGI 脚本又是如何将处理好的房间 1 的灯光设备列表数据发送给客户端浏览器的呢？这就是本小节讨论的重点。

1. 客户端传输数据的获取

事实上，当服务器守护进程接收到客户端用户代理（如浏览器）提交的 CGI 请求时，所创建的 CGI 子进程会设置与 CGI 请求内容有关的环境变量，并建立服务器与外部 CGI 程序之间通信的通道（即标准 I/O）。CGI 程序可以通过环境变量、标准 I/O 或命令行参数获取客户端用户输入的数据。客户端传输数据的具体获取方法和用户所使用的请求方式（GET 或 POST 方法）有关。

1) 获取环境变量

环境变量的类别很多，包含客户端和服务器的详细信息。在一般的 CGI 程序开发中，下述几个环境变量在数据传递中起着重要作用。

(1) GATEWAY-INTERFACE: CGI 程序所使用的 CGI 标准接口的版本号。如使用 CGI1.1 版，该变量表示为“CGI/1.1”。

(2) REQUEST-METHOD: HTTP 请求方法。根据该变量值可判断 CGI 请求所采用的请求方法，以决定是通过 stdin 还是通过环境变量 QUERY-STRING 获取客户端传输数据。

(3) QUERY-STRING: QUERY-STRING 是 CGI 程序 URL 中“?”之后的数据。当使用 ISINDEX 查询或 FORM 表使用 GET 方法时，客户端传输数据可以通过读取该变量而获得。

(4) CONTENT-LENGTH: CONTENT-LENGTH 表示客户端传输数据的字节数。

(5) CONTENT-TYPE: CONTENT-TYPE 表示客户端传输数据的数据编码类型。

利用 environ(int)函数可以获得所有环境变量及其值；利用 getenv(constchar*)函数可以获得指定环境变量的相应值。

2) HTTP 请求方法

客户端用户代理提交的 CGI 请求是 HTTP 请求，其中包括 HTTP 请求方法。HTTP 协议

定义的请求方法中常用的主要有 GET 和 POST。

如果客户端使用 GET 方法, CGI 程序通过环境变量 QUERY-STRING 获取客户端传输数据; 如果客户端使用 POST 方法, 则 CGI 程序通过 CONTENT-LENGTH 获取客户端传输数据的字节数, 通过 stdin 读取客户端传输数据。

2. 有效数据的提取和处理

通过上述方法可以获取 CGI 请求中“?”以后的字符串, 该字符串的一般形式为 name[1]=value[1]&name[2]=value[2]&...&name[n]=value[n], 其中 name[i]表示变量名, value[i]表示变量值。很显然, 只要通过适当的字符串处理函数(如 strtok 函数), 就可以依次分离出各个键值对字符串数组。

3. 向客户端返回应答

CGI 程序处理结束后, 通过标准输出流将应答信息传递给服务器, 再由服务器返回给发出请求的客户端。其输出的应答信息是 HTTP 应答消息, 它一般由两部分组成: 应答头和应答数据。

常见的应答头包括三种头域: Content-Type (数据编码类型, 用 MIME 类型表示)、Location (特定文档的 URL, 这种情况不直接向客户端输出内容而输出该 URL) 和 Status (处理结果的状态码和状态描述)。HTTP 应答头由几行格式相同的文本构成, 每一行的基本格式为“头域名:该域内容”。应答头和应答体之间用一个空行分隔。应答体为 CGI 扩展程序的输出数据, 其数据类型应该与 Content-Type 值相一致。

CGI 程序的输出可以用 printf()、puts()等标准 I/O 函数来实现。

4. 程序实现

为了实现上述提取 CGI 请求数据的方法, 相应的程序如下:

```
/**将双字十六进制字符串转换为单个 ASCII 字符**/
char x2c(char*what) {
    register char digit;
    digit = (what[0] >= 'A' ? ((what[0] & 0xdf) - 'A') + 10 : (what[0] - '0'));
    digit *= 16;
    digit += (what[1] >= 'A' ? ((what[1] & 0xdf) - 'A') + 10 : (what[1] - '0'));
    return(digit);
}

/**将经过编码而发生转义的字符串转换为常规 ASCII 字符串**/
void unescape_url(char*url) {
    register int i,j;
    for(i=0,j=0; url[j]; ++i,++j) {
        if((url[i] = url[j]) == '%') {
            url[i] = x2c(&url[j+1]);
            j += 2;
        }
    }
    url[i] = '\0';
}
```

```

/**读取 CGI 请求的字符串，并返回包含所有键值对字符串的字符串数组**/
/**返回的字符串数组包含 name1, value1, name2, value2, ... , NULL**/
char**getcgivars() {
    register int i ;
    char*request_method ;
    int content_length;
    char*cgiinput ;
    char**cgivars ;
    char**pairlist ;
    int paircount ;
    char*nvpair ;
    char*eqpos ;

    /**根据请求方式，读取所有的 CGI 请求字符串到 cgiinput 中**/
    request_method= getenv("REQUEST_METHOD") ;
    //GET 方式
    if (!strcmp(request_method, "GET") || !strcmp(request_method, "HEAD")) {
        /*如果请求字符串为空，一些服务器不会提供 QUERY_STRING 环境变量，所以要避免
getenv("QUERY_STRING")返回 NULL 的情况*/
        char*qs ;
        qs= getenv("QUERY_STRING") ;    //从 QUERY_STRING 环境变量获取请求字符串
        cgiinput= strdup(qs ? qs : "");
    }
    else if (!strcmp(request_method, "POST")) { //POST 方式
        if ( strcmp(getenv("CONTENT_TYPE"), "application/x-www-form-urlencoded")) {
            printf("Content-Type: text/plain\n\n") ;
            printf("getcgivars(): Unsupported Content-Type.\n") ;
            exit(1) ;
        }
        if ( !(content_length = atoi(getenv("CONTENT_LENGTH"))) ) {
            printf("Content-Type: text/plain\n\n") ;
            printf("getcgivars(): No Content-Length was sent with the POST request.\n") ;
            exit(1) ;
        }
        if ( !(cgiinput= (char*) malloc(content_length+1)) ) {
            printf("Content-Type: text/plain\n\n") ;
            printf("getcgivars(): Couldn't malloc for cgiinput.\n") ;
            exit(1) ;
        }
        if (!fread(cgiinput, content_length, 1, stdin)) { //从 stdin（标准输入）读入数据
            printf("Content-Type: text/plain\n\n") ;
            printf("getcgivars(): Couldn't read CGI input from STDIN.\n") ;
            exit(1) ;
        }
        cgiinput[content_length]='\0' ;
    }
}

```

```

    }
    else {
        printf("Content-Type: text/plain\n\n");
        printf("getcgivars(): Unsupported REQUEST_METHOD.\n");
        exit(1);
    }

    /**将所有的“+”变成空格**/
    for (i=0; cgiinput[i]; i++) if (cgiinput[i] == '+') cgiinput[i] = ' ';

    /**首先, 根据“&”和“;”分离成 name-value 字符串数组保存到 pairlist[]中**/
    pairlist= (char**) malloc(256*sizeof(char**));
    paircount= 0;
    nvpair= strtok(cgiinput, "&;");
    while (nvpair) {
        pairlist[paircount++]= strdup(nvpair);
        if (!(paircount%256))
            pairlist= (char**) realloc(pairlist, (paircount+256)*sizeof(char**));
        nvpair= strtok(NULL, "&;");
    }
    pairlist[paircount]= 0;          /*以 NULL 作为字符串数组结尾*/

    /**然后, 从 pairlis[]中准确分离出每一个 name 和 value 字符串保存到 cgivars[]中**/
    cgivars= (char**) malloc((paircount*2+1)*sizeof(char**));
    for (i= 0; i<paircount; i++) {
        if (eqpos=strchr(pairlist[i], '=')) {
            *eqpos= '\0';
            unescape_url(cgivars[i*2+1]= strdup(eqpos+1));
        } else {
            unescape_url(cgivars[i*2+1]= strdup(""));
        }
        unescape_url(cgivars[i*2]= strdup(pairlist[i]));
    }
    cgivars[paircount*2]= 0;        /*以 NULL 作为字符串数组结尾*/

    /**释放内存空间**/
    free(cgiinput);
    for (i=0; pairlist[i]; i++) free(pairlist[i]);
    free(pairlist);

    /**返回 cgivars[]字符串数组**/
    return cgivars;
}

```

至此, 在 CGI 脚本程序中只需简单地调用 `getcgivars()` 函数, 就能获取处理好的包含变

量名和变量值的字符串数组。

9.7.2 程序中读写 SQLite 数据库

SQLite 数据库作为网关数据的组织、存储容器，后台 CGI 脚本需要经常对其进行读写等操作。本小节的讨论重点就是在程序中如何读写 SQLite 数据库。

如果要在程序中对其他的传统大型数据库系统（如 MS SQL Server、MySQL 等）进行操纵，那将是一件很麻烦的事情。幸运的是，SQLite 作为一款以“简单”为目标的轻型数据库，在程序中我们可以通过其提供的 API 非常容易地对 SQLite 数据库进行读写操作。

如前所述，在编译成功后生成的 3 个文件夹中，include/文件夹中包含了两个头文件——sqlite3.h 和 sqlite3ext.h，其中 sqlite3.h 是我们讨论的重点，所有编程中需要涉及的 API 函数的定义都包含在该头文件中。

SQLite 3.0 一共有 83 个 API 函数，此外还有一些数据结构和预定义。虽然数量不少，但是在一般开发过程中常用的却不多，最简单的程序只需使用 3 个函数就可以完成：sqlite3_open()、sqlite3_exec()和 sqlite3_close()。如果想更好地控制数据库引擎的执行，可以使用提供的 sqlite3_prepare()函数把 SQL 语句编译成字节码，然后再使用 sqlite3_step()函数来执行编译后的字节码。以 sqlite3_column_开头的一组 API 函数用来获取查询结果集中的信息。许多接口函数都是成对出现的，同时有 UTF-8 和 UTF-16 两个版本，并且提供一组函数用来执行用户自定义的 SQL 函数和文本排序函数。下面仅对在实际项目开发过程中比较重要、常用的 API 函数做详细介绍。

1. sqlite3_open

打开一个 SQLite 数据库。

```
int sqlite3_open(  
    const char*filename,          /*Database filename (UTF-8)*/  
    sqlite3**ppDB                 /*OUT: SQLite db handle*/  
);
```

参数：filename——包含要打开的数据库的路径和文件名的字符串。

ppDB——为一个 sqlite3 结构体指针，该结构体用于表示一个打开的 SQLite 数据库对象，在后续的数据库操作中均需要使用。

返回值：打开成功返回 SQLITE_OK（#define SQLITE_OK 0）；反之，则返回相应的错误代码，可使用 sqlite3_errmsg()函数获取错误描述。

说明：如果指定的数据库不存在，则先创建，然后再打开。打开或者创建数据库的命令会被缓存，直到这个数据库真正被调用时才会执行。

2. sqlite3_close

关闭一个 SQLite 数据库连接。

```
int sqlite3_close(sqlite3*pDB);
```

参数：pDB——要关闭的数据库的 sqlite3 结构体指针。

返回值：关闭成功返回 SQLITE_OK；反之，则返回相应的错误代码，可使用

sqlite3_errmsg()函数获取错误描述。

3. sqlite3_errcode

获取最近一次数据库操作的错误代码。

```
int sqlite3_errcode(sqlite3*db);
```

参数：db——要获取错误代码的 SQLite 数据库结构体对象。

返回值：该数据库最近一次操作失败的错误代码。

说明：该函数只能获取最近的操作失败错误代码，如果前一次的操作失败，但最近一次的操作却是成功的，则该函数的返回值将是不确定的。

4. sqlite3_errmsg

获取最近一次数据库操作的错误描述文本。

```
const char*sqlite3_errmsg(sqlite3*db);
```

参数：db——要获取错误描述的 SQLite 数据库结构体对象。

返回值：包含该数据库最近一次操作失败错误描述的字符串。

说明：该函数返回的字符串描述为英文描述，并且在下一次针对该数据库的操作完毕后将被改写。

5. sqlite3_exec

执行单步（One-step）SQL 语句。

```
int sqlite3_exec(
    sqlite3*db,                /*An open database*/
    const char*sql,            /*SQL to be evaluated*/
    int (*callback)(void*,int,char**,char**), /*Callback function*/
    void*arg,                 /*1st argument to callback*/
    char**errmsg);             /*Error msg written here*/
```

参数：db——要执行的数据库的 sqlite3 结构体对象指针。

sql——要执行的 SQL 语句字符串。

callback——SQL 语句执行完毕后调用的回调函数指针。回调函数形如 int Func (void*,int,char**,char**)。

arg——传递给回调函数的第一个参数值。

errmsg——错误描述字符串。

返回值：执行成功返回 SQLITE_OK；反之，则返回相应的错误代码，可使用 sqlite3_errmsg()函数获取错误描述。

说明：

(1) 第二个参数 SQL 语句若为空，则不会对数据库做出任何更改。

(2) 第三个参数回调函数是可选的，对于某些不需要返回值的 SQL 查询语句，如 INSERT、UPDATE 等，该参数可设置成 0。

(3) 参数 errmsg 返回的错误描述字符串由 sqlite3_malloc()函数分配内存，为避免内存

泄露，在程序尾部应使用 `sqlite3_free()` 函数来进行内存释放。

6. `sqlite3_get_table`

执行 SQL 语句，并获取完整的执行结果表（包含所有查询结果数据的二维字符串数组）。

```
int sqlite3_get_table(
    sqlite3*db,           /*An open database*/
    const char*zSql,      /*SQL to be evaluated*/
    char***pazResult,     /*Results of the query*/
    int*pnRow,            /*Number of result rows written here*/
    int*pnColumn,         /*Number of result columns written here*/
    char**pzErrMsg);      /*Error msg written here*/
```

参数：db——进行 SQL 查询的数据库 `sqlite3` 结构体对象指针。

zSql——包含要执行的 SQL 语句的字符串。

pazResult——包含所有查询结果的字符串二维数组指针。

pnRow——查询结果中包含的行数。

pnColumn——查询结果中包含的列数。

pzErrMsg——错误描述字符串。

返回值：执行成功返回 `SQLITE_OK`；反之，则返回相应的错误代码，可使用 `sqlite3_errmsg()` 函数获取错误描述。

说明：返回的结果表（二维字符串数组）pazResult 由 `sqlite3_get_table` 负责初始化内存，并用查询结果填充，故用户只需提供二维字符串数组指针即可，而且程序结尾应使用 `sqlite3_free_table()` 函数进行释放，以避免内存泄露。

函数返回的结果表实质上是一个包含 UTF-8 字符串的字符串数组，它包含 $(N+1) \cdot M$ （ N 为查询结果的实际行动数， M 为查询结果的实际行动列数）个元素，结果表的第一行为所查询的列名。例如，查询得到的结果表如下：

Name	Age
Alice	43
Bob	28
Cindy	21

假设该表存放在 azResult 中，则 azResult 中的内容如下：

```
azResult[0] = "Name";      //第一列列名
azResult[1] = "Age";       //第二列列名
azResult[2] = "Alice";
azResult[3] = "43";
azResult[4] = "Bob";
azResult[5] = "28";
azResult[6] = "Cindy";
```

```
azResult[7] = "21";
```

7. sqlite3_free_table

释放结果表占用的内存空间。

```
void sqlite3_free_table(char**result);
```

参数：result——要释放的结果表（即字符串二维数组）指针。

说明：该函数只能用于释放通过 sqlite3_get_table() 函数获取的结果表，要释放其他 sqlite3 对象请使用 sqlite3_free() 函数。

8. sqlite3_free

释放 sqlite3 对象占用的内存空间。

```
void sqlite3_free(void*obj);
```

参数：obj——要释放的 sqlite3 对象指针。

说明：该函数可用于释放通过 sqlite3_malloc() 和 sqlite3_realloc() 函数申请的内存空间。即使传入的对象指针为 NULL，函数也不会发生错误。

以上提及的这些 SQLite API 对于普通的编程应用已完全足够。

9.7.3 RS-485 串口读写

如前节所述，RS-485 也为串口范畴，只是接口标准和传输方式有所不同，所以对 RS-485 的读写和对普通的 RS-232 串口读写在本质上没有任何区别。下面着重讨论 Linux 下对串口进行设置和读写的方法。

1. 串口操作

串口操作需要的头文件如下：

```
#include <stdio.h>           /*标准输入/输出定义*/
#include <stdlib.h>          /*标准函数库定义*/
#include <unistd.h>          /*UNIX 标准函数定义*/
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>           /*文件控制定义*/
#include <termios.h>         /*PPSIX 终端控制定义*/
#include <errno.h>           /*错误号定义*/
```

2. 打开串口

在 Linux 中串口文件是位于 /dev 下的，串口 1 为 /dev/ttyS0，网关所使用的 RS-485 端口为 /dev/ttyS1。

打开串口是通过使用标准的文件打开函数操作的。以下为打开串口的函数实现：

```
/**
 * @brief 打开指定串口
 * @param Dev 需要打开的串口文件名（如"/dev/ttyS0"）
```

```

    *@return 如果打开成功，返回打开的串口句柄；如果失败则返回-1
    */
    int sc_openport(char*Dev)
    {
        int    fd = open( Dev, O_RDWR );           //打开指定串口
        if (-1 == fd)
        {
            perror("Can't Open Serial Port");
            return -1;
        }
        return fd;
    }

```

3. 串口设置

最基本的串口设置包括波特率设置、校验位和停止位设置。串口的设置主要是设置 `termios` 结构体的各成员值，该结构体定义如下：

```

struct termio
{
    unsigned short  c_iflag;           /*输入模式标志*/
    unsigned short  c_oflag;           /*输出模式标志*/
    unsigned short  c_cflag;           /*控制模式标志*/
    unsigned short  c_lflag;           /*local mode flags*/
    unsigned char   c_line;             /*line discipline*/
    unsigned char   c_cc[NCC];         /*control characters*/
};

```

设置这个结构体很复杂，我们只讨论一些常见的设置。

1) 波特率设置

下面是设置串口波特率的函数实现：

```

/**
 *@brief 设置串口通信速率
 *@param fd    类型 int  打开串口的文件句柄
 *@param speed 类型 int  串口速度
 *@return void
 */
int speed_arr[] = { B38400, B19200, B9600, B4800, B2400, B1200, B300,
                    B38400, B19200, B9600, B4800, B2400, B1200, B300, };
int name_arr[] = { 38400,  19200,  9600,  4800,  2400,  1200,  300, 38400,
                   19200,  9600, 4800, 2400, 1200,  300, };
void sc_setspeed(int fd, int speed){
    int    i;
    int    status;
    struct termios  Opt;           /*声明 termios 结构体对象*/
    tcgetattr(fd, &Opt);          /*获取串口默认设置*/
    for ( i= 0;  i < sizeof(speed_arr) / sizeof(int);  i++) {

```

```

        if (speed == name_arr[i]) {
            tcflush(fd, TCIOFLUSH);          /*清空串口输入/输出缓存*/
            cfsetispeed(&Opt, speed_arr[i]);  /*设置输入波特率*/
            cfsetospeed(&Opt, speed_arr[i]);  /*设置输出波特率*/
            status = tcsetattr(fd1, TCSANOW, &Opt); /*设置新属性*/
            if (status != 0) {
                perror("tcsetattr fd1");
                return;
            }
            tcflush(fd, TCIOFLUSH);          /*清空串口输入/输出缓存*/
        }
    }
}

```

2) 校验位和停止位设置（见表 9-1）

表 9-1 串口的校验位和停止位设置

无校验	8 位	Option.c_cflag &= ~PARENB; Option.c_cflag &= ~CSTOPB; Option.c_cflag &= ~CSIZE; Option.c_cflag = ~CS8;
奇校验 (Odd)	7 位	Option.c_cflag = ~PARENB; Option.c_cflag &= ~PARODD; Option.c_cflag &= ~CSTOPB; Option.c_cflag &= ~CSIZE; Option.c_cflag = ~CS7;
偶校验 (Even)	7 位	Option.c_cflag &= ~PARENB; Option.c_cflag = ~PARODD; Option.c_cflag &= ~CSTOPB; Option.c_cflag &= ~CSIZE; Option.c_cflag = ~CS7;
Space 校验	7 位	Option.c_cflag &= ~PARENB; Option.c_cflag &= ~CSTOPB; Option.c_cflag &= &~CSIZE; Option.c_cflag = CS8;

下面是设置串口校验位和停止位的函数实现：

```

/**
 * @brief 设置串口数据位、停止位和校验位
 * @param fd 类型 int 打开的串口文件句柄
 * @param databits 类型 int 数据位 取值为 7 或者 8
 * @param stopbits 类型 int 停止位 取值为 1 或者 2
 * @param parity 类型 int 效验类型 取值为 N,E,O,S
 */
int sc_setparity(int fd, int databits, int stopbits, int parity) {
    struct termios opt;
    if(tcgetattr(fd, &opt) != 0) { /*得到设备当前模式*/
        perror("setup serial...");
        return 0;
    }
}

```

```

    }
    opt.c_cflag &= ~CSIZE; /*设置数据位数前，首先使用“CSIZE”屏蔽数据位数*/
    opt.c_oflag&=~OPOST; /*使用原始模式，禁止 Linux 对串口发送数据 0x0a 和 0x0d 进行
转义。不处理输出，即忽略所有其他输出选项*/
    switch(databits) { /*设置数据位数*/
        case 7:
            opt.c_cflag |= CS7; /*设置数据位数=7*/
            break;
        case 8:
            opt.c_cflag |= CS8; /*设置数据位数=8*/
            break;
        default:
            printf("unsupport data size.\n");
            return 0;
    }
    switch(parity) { /*设置串口校验位*/
        case 'n':
        case 'N':
            opt.c_cflag &= ~PARENB; /*设置校验位为 0，禁止校验*/
            opt.c_iflag &= ~INPCK;
            break;
        case 'o':
        case 'O':
            opt.c_cflag |= (PARENB|PARODD);
            opt.c_iflag |= INPCK;
            break;
        case 'e':
        case 'E':
            opt.c_cflag |= PARENB;
            opt.c_cflag &= ~PARODD;
            opt.c_iflag |= INPCK;
            break;
        case 's':
        case 'S':
            opt.c_cflag &= ~PARENB;
            opt.c_cflag &= ~CSTOPB; /*设置双停止位*/
            break;
        default:
            printf("unsupported parity.\n");
            return 0;
    }
    switch(stopbits) { /*设置停止位*/
        case 1:
            opt.c_cflag &= ~CSTOPB;
            break;
    }

```

```

        case 2:
            opt.c_cflag &= CSTOPB;
            break;
        default:
            printf("unsupported stopbits.\n");
            return 0;
    }

    if(parity != 'n') opt.c_iflag |= INPCK;
    tcflush(fd,TCIFLUSH);
    opt.c_cc[VTIME] = 150;
    opt.c_cc[VMIN] = 0;

    if(tcsetattr(fd,TCSANOW,&opt) != 0) {
        perror("\n");
        return 0;
    }
    return 1;
}

```

4. 读写串口

设置好串口之后，读写串口就很容易了，把串口当作文件读写即可。

1) 发送数据

```

char buffer[1024];
int Length;
int nByte;
nByte = write(fd, buffer ,Length);

```

2) 读取串口数据

使用文件操作 `read` 函数读取串口数据，如果设置为原始模式传输数据，则 `read` 函数返回的字符数是串口实际收到的字符数。可以使用操作文件的函数来实现异步读取，如 `fcntl` 或 `select` 等。

```

char buff[1024];
int Len;
int readByte = read(fd,buff,Len);

```

5. 关闭串口

关闭串口就是关闭文件。

```

close(fd);

```

9.7.4 指令格式定义

1. 网关指令

网关通过 RS-485 串口向所有的转发器广播发送特定的指令，转发器在收到属于自己的

指令后进行适当的处理，然后将指令通过 RF 或者红外的方式发射出去，从而实现对遥控开关或家电的控制。

可见，在网关和转发器之间需要定义一个合理的指令格式，这样每个转发器才能正确判断出当前指令是否由自己负责发射并区分指令的类别等。为此，我们规定网关发送指令必须遵循以下格式：

BYTE4	C31 C30 C29 C28 C27 C26 C25 C24	
BYTE3	C23 C22 C21 C20 C19 C18 C17 C16	
BYTE2	C15 C14 C13 C12 C11 C10 C9 C8	
BYTE1	C7 C6 C5 C4 C3 C2 C1 C0	/*C 为遥控指令，长为 4 字节（32 位）*/
BYTE0	A3 A2 A1 A0 P 1 1 0	/*P 为指令类型标志位；A 为转发器编号，4 位，支持 16 个转发器*/

发送顺序：低位先发，如 0 1 1 P A0 A1 A2 A3 C0 C1 ... C31

对于以上指令格式的说明如下。

(1) BYTE0 低 3 位固定为 1 1 0，一方面是为了和稍后将要介绍的遥控开关指令有所区别，因为遥控开关指令前 3 位是 1 1 0 时为保留状态；另一方面，作为指令头，方便转发器对接收到的指令做出判断和识别。

(2) P 为指令类型标志位，该位为 0 表示为家电控制遥控指令，该位为 1 表示为安防报警指令。

(3) C 部分（BYTE1~BYTE4）为遥控开关 RF 指令或家电红外遥控指令，该部分固定为 4 字节长度，指令总长度（011+P+A+C）为 5 字节。

2. 遥控开关指令

因为遥控开关厂商指定了自己的指令格式，所以这里有必要对遥控开关的指令格式做一个简要介绍。

1) 发码指令组成

发码有效数据的总长度为 19b，配置位为 1b，功能位为 2b，数据位为 4b，地址位为 12b。格式定义如下：

BYTE2 (x x x x A11 A10 A9)	/*x 为空，K 为数据位*/
BYTE1 (A8 A7 A6 A5 A4 A3 A2 A1)	/*A 为地址位*/
BYTE0 (A0 K3 K2 K1 K0 S1 S0 N)	/*K 为数据位，S 为功能位，N 为配置位*/

发码顺序：配置位+功能位+数据位+地址位 功能位/数据位/地址位，都是低位先发，即 NS0S1K0K1K2K3A0A1 A2 A3 ... A11

注意：因为遥控开关指令将嵌入前面介绍的网关指令中，所以这里所指的 BYTE0~BYTE2，实际上为网关指令中的 BYTE1~BYTE3。

2) 指令格式说明

(1) 配置位：

- 当配置位为 0 时，其后面的功能位和数据位表示按键；

- 当配置位为 1 时，其后面的功能位和数据位表示命令或情景。
- (2) 功能位：
- 对数据位的含义进行附加说明。
- 当配置位为 0 时，对应按键功能，如下所示：

含 义	值 (S1S0)	备 注
乒乓按键	00	
Only 关	01	
Only 开	10	
reserved	11	

- 当配置位为 1 时，对应命令/情景功能，如下所示：

含 义	值 (S1S0)	备 注
情景	00	
调光减	01	
调光加	10	
命令	11	其后为数据位和地址位，不全为 1
工厂自检	11	其后为数据位和地址位，均为 1

- (3) 数据位：
- 配置位为 0 时，4 位数据位可表示 16 个按键。
 - 配置位为 1, 功能位 S1S0 为 00 时，可表示 16 个情景，如下所示：

含 义	值 (K3K2K1K0)	备 注
全开	0000	
全关	0001	
...		保留
自定 1	1000	
自定 2	1001	
自定 3	1010	
自定 4	1011	
自定 5	1100	
自定 6	1101	
...		

- 配置位为 1，功能位 S1S0 为 11 时，可表示 16 个命令，如下所示：

情景学习/修改	0000	(不适合于全开/全关两个情景)
reserved	0010	
reserved	0011	
reserved	0100	

(4) 地址位: 12 位地址位, 即 $2^{12} = 4096$ 个遥控器不冲突。在使用时可以增加此地址编码的位数, 以扩充设备或进行其他功能设定。

9.7.5 CGI 脚本举例

关于 CGI 程序设计与实现的讨论已临近尾声, 到目前为止, 我们已经探讨了编写网关后台 CGI 脚本程序所需的所有技术与指令格式。在这一节中, 将从网关后台 CGI 脚本中选取一个最典型的脚本进行深入的讲解。

将要讲解的这个 CGI 脚本的主要作用为: 根据用户的选择, 发送特定的情景指令, 控制所有进行过情景学习的遥控开关进入相应的情景状态。

客户端的 CGI 请求形如 “cgi-bin/scenecmd.cgi?roomid=1&cmdtype=1”。

接收到 CGI 请求后, 该脚本首先提取 CGI 请求中的房间号和要发送的情景状态标识号。因为一个房间中完全可能有多个转发器, 并且每个转发器可能需充当多个不同的智能开关遥控器, 所以根据获取的房间号, 必须从 SQLite 数据库的 switchinfo 表中查询属于该房间的所有转发器编号和遥控器地址, 然后根据每一条查询到的记录逐一通过 RS-485 串口发送指定的情景指令。

以上就是该 CGI 脚本的大致业务流程, 下面是该 CGI 脚本的程序实现:

```
/*scenecmd.c 文件*/
#include "stdlib.h"
#include "stdio.h"
#include "string.h"
#include "../commoncodes/sqlite3.h"    //sqlite3 头文件
#include "../commoncodes/rs485.h"      //rs485 头文件, 封装了各串口函数
const char*dbpath="../sqlite/gateway.db"; //数据库文件路径
extern char**getcgivars(); /*声明外部函数 getcgivars(), 该函数位于文件 httpcgiparse.c 中*/

int main()
{
    int i,rc,nRow,nColumn,routeid=0,swid=0;
    char **cgivars,*sql,**pResult,sendcmd[5];
    sqlite3*pDB=NULL;                //sqlite3 结构体对象

    rc=sqlite3_open(dbpath,&pDB);      //打开数据库文件
    if (rc) return 1;                 //如果打开失败, 返回

    cgivars=getcgivars();              //获取 CGI 请求中的数据
    printf("Content-Type: text/plain\n\n"); //向客户端发送应答头信息

    sql=(char*)malloc(100);
    sprintf(sql,"select distinct routeid,swid from switchinfo where roomid=%d",atoi(cgivars[1])); /*组
    建 SQL 语句, 根据 CGI 请求中的房间号, 在 switchinfo 表中查询转发器编号和遥控器地址*/
    rc=sqlite3_get_table(pDB,sql,&pResult,&nRow,&nColumn,NULL); /*执行 SQL 语句, 并获取
```

```

结果表 pResult*/
    if (rc) return 1;                                //如果执行失败，返回

    if (!rs485_Init()) return 1;                     //初始化 RS-485 端口
    for(i=2;i<(nRow+1)*nColumn;i+=2) //因第一行为列名，故从第3个元素开始
    {
        routeid=atoi(pResult[i]);                 //获取转发器编号
        swid=atoi(pResult[i+1]);                   //获取遥控器地址
        /*组建将要发送的指令*/
        memset(sendcmd,0,sizeof(sendcmd));
        sendcmd[0]=0x06;                            //指令校验头 0 1 1 0
        sendcmd[0]=routeid<<4;                       //转发器编号
        switch (atoi(cgivars[3]))                   //根据请求中的情景标识号组建相应指令
        {
            case 0:                                   //情景学习 0000 0111
                sendcmd[1]=0x07;
                break;
            case 1:                                   //全开 0000 0001
                sendcmd[1]=0x01;
                break;
            ...
            default: return 1;
        }
        //根据指令格式定义，填充开关地址
        sendcmd[1]=swid<<7;
        sendcmd[2]=swid>>1;
        sendcmd[3]=swid>>9;

        rs485_Send(sendcmd,sizeof(sendcmd),3,200) //通过 RS-485 发送指令
        usleep(100);                               //暂停 100μs 后继续发送下一条指令
    }

    rs485_Close();                                  //关闭 RS-485
    sqlite3_close(pDB);                             //关闭打开的数据库连接
    free(sql);                                       //释放分配的内存空间
    sqlite3_free_table(pResult);                    //释放 pResult 结果表占用的内存空间
    for (i=0;cgivars[i];i++) free(cgivars[i]);      //释放 cgivars[] 数组占用的空间
    free(cgivars);

    printf("success=true;");//发送 JavaScript 语句，告诉客户端 CGI 执行成功
    return 0;
}

```

该 CGI 脚本源代码中的注释已十分详尽，这里不再赘述。

需要说明的是：函数 rs485_Init()、rs485_Send()和 rs485_Close()对串口函数进行了封

装，分别为初始化、发送和关闭 RS-485 串口。其中，rs485_Send()函数具有在指定时间间隔内重发指定次数指令的功能，在此例中，我们让 RS-485 每条指令重发 3 次，每次间隔为 200ms，为的是避免转发器接收指令时遇到错误。

此外，在程序最后发送的“success=true;”为一句 JavaScript 语句。在 JavaScript 中，可以使用 eval()方法将接收到的文本作为 JavaScript 语句来执行。在这里，我们先在 JavaScript 脚本中定义“success=false;”，然后用 eval()方法执行接收到的文本，执行后如果 success==true 就说明 CGI 执行成功，反之则 CGI 执行失败。

至此，关于 CGI 脚本程序的设计与实现的探讨告一段落。下一节开始将着重探讨前台 Web 的设计与实现方法。

9.8 前台网页设计与实现

前台 Web 页面作为用户访问网关并进行相应控制、设置的唯一途径，对其的设计和开发工作显得尤为重要。

本节主要讨论前台 Web 页面的设计与实现方法。一个优美的、具有交互性的网页必定由 HTML、JavaScript 和 CSS 文件组成，同时使用 Ajax 技术与服务器进行通信，给用户一种全新的用户体验。

9.8.1 HTML 简介

HTML (HyperText Mark-up Language) 即超文本标记语言或超文本链接标示语言，是目前网络上应用最为广泛的语言，也是构成网页文档的主要语言。设计 HTML 语言的目的是为了把存放在一台计算机中的文本或图形与另一台计算机中的文本或图形方便地联系在一起，形成有机的整体，人们不用考虑具体信息是在当前计算机上还是在网络的其他计算机上。用户只需使用鼠标在某一文档中点取一个图标，Internet 就会立即转到与此图标相关的内容上去，而这些信息可能存放在网络的另一台计算机中。HTML 文本是由 HTML 命令组成的描述性文本，HTML 命令可以说明文字、图形、动画、声音、表格、链接等。HTML 的结构包括头部 (Head) 和主体 (Body) 两大部分，其中头部描述浏览器所需的信息，主体则包含所要说明的具体内容。

另外，HTML 是网络的通用语言，一种简单、通用的标记语言。它允许网页制作人建立文本与图片相结合的复杂页面，这些页面可以被网上的任何人浏览到，无论其使用的是何种类型的计算机或浏览器。

事实上，HTML 只不过是组合成一个文本文件的一系列标签。HTML 标签通常是英文词汇的全称 (如块引用标签“blockquote”) 或缩略语 (如“p”标签代表 Paragraph)，但它们与一般的文本有区别，因为它们放在单书名号里。故 Paragraph 标签是<p>，块引用标签是<blockquote>。有些标签说明页面如何被格式化 (如开始一个新段落)，其他则说明这些内容是如何显示的 (如使文字变粗)，还有一些标签提供在页面上不显示的信息 (如标题)。

关于标签，需要记住的是，它们是成对出现的。每当使用一个标签，如<blockquote>

时，则必须以另一个标签</blockquote>将它关闭。注意，“blockquote”前的斜杠，那就是关闭标签与打开标签的区别。但是也有一些标签例外，比如，<input>标签。

基本 HTML 页面以<html>标签开始，以</html>结束。在它们之间，整个页面有两部分内容——标题和正文。

标题词位于<head>和</head>标签之间，这个词语在打开页面时出现在屏幕底部最小化的窗口中。正文则位于<body>和</body>之间，即所有页面的内容所在。页面上显示的任何东西都包含在这两个标签之中。

由于完整的 HTML 规范较为庞大、繁杂，所以这里不可能、也没必要进行详细阐述。下面，我们将从家庭网关所有的 HTML 页面文件中挑选一个比较典型的 HTML 页面进行解析。

为了延续前面介绍内容中举例的 CGI 脚本，我们选取 SceneCtrl.html 这个页面进行讲解，该页面的用途是进行情景控制。

SceneCtrl.html 页面的 HTML 代码如下：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <!--HTML 标题-->
    <title>家庭网关->家电控制->情景控制</title>
  </head>
  <body>
    <h1>情景控制</h1>
    <!--此处用于显示当前房间名，将在讨论 JavaScript 时具体讨论-->
    <h2><span id="roomname"></span></h2>
    <div>
      <!--情景指令表-->
      <table id="cmdtable">
        <tr>
          <td id="cmd1">全开</td>
          <td id="cmd2">全关</td>
          <td id="cmd3">起夜</td>
        </tr>
        <tr>
          <td id="cmd4">会客</td>
          <td id="cmd5">娱乐</td>
          <td id="cmd6">自定</td>
        </tr>
      </table>
    </div>
    <div id="btns">
      <!--点此按钮返回上一页 homecontrol.html-->
```

```

        <button onclick="window.location.href='homecontrol.html'">退出</button>
    </div>
</body>
</html>

```

因为目前还未给该 HTML 添加 CSS 样式表，所以 HTML 布局看上去比较混乱。目前该 HTML 页面的效果如图 9-9 所示。

没有 CSS 的布局 and 美化，该网页显得很枯燥、零乱。在后面我们将为该页面加上 CSS 样式表，使页面看上去更人性化。

9.8.2 CSS 简介

CSS 的全称是 Cascading Style Sheets（层叠样式表）。CSS 语言是一种标记语言，它不需要编译，可以直接由浏览器执行（属于浏览器解释型语言）。在标准网页设计中，CSS 负责网页内容（XHTML）的表现。CSS 文件也可以说是一个文本文件，它包含了一些 CSS 标记，CSS 文件必须使用 css 为文件名后缀。可以通过简单地更改 CSS 文件，从而改变网页的整体表现形式，以减小网页设计者的工作量，所以 CSS 是每一个网页设计人员必须掌握的语言之一。

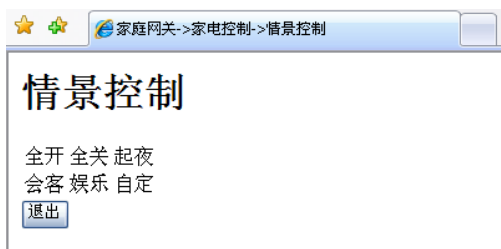


图 9-9 未添加 CSS 样式表的 HTML 页面

1. CSS 的语法

CSS 的定义由三个部分构成：选择符（selector）、属性（property）和属性的取值（value）。

其语法形式为：

```
selector {property: value} （选择符 {属性:值}）
```

选择符可以是多种形式，一般为要定义样式的 HTML 标记，如 BODY、P、TABLE 等。设计者可以通过此方法定义它的属性和值，属性和值要用冒号隔开，例如：

```
body {color: black} /*此例的效果是使页面中的文字为黑色*/
```

如果属性的值由多个单词组成，则必须在值上加引号。例如，字体的名称经常是几个单词的组合，则有：

```
p {font-family: "sans serif"} /*定义段落字体为 sans serif*/
```

当需要对一个选择符指定多个属性时，使用分号将所有的属性和值分开，例如：

```
p {text-align: center; color: red} /*段落居中排列，并且段落中的文字为红色*/
```

可以把相同属性和值的选择符组合起来书写，用逗号将选择符分开，这样可以减少样式重复定义，例如：

```
p, table{ font-size: 9pt } /*段落和表格里文字尺寸为 9 号字*/
```

其效果完全等同于：

```
p { font-size: 9pt }
table { font-size: 9pt }
```

2. 在 HTML 中加入 CSS 的方法

可以用以下三种方式将样式表加入网页中。越接近目标的样式定义优先权越高。高优先权样式将继承低优先权样式的未重叠定义但覆盖重叠的定义。

1) 链入外部样式表文件 (Linking to a Style Sheet)

可以先建立外部样式表文件 (.css)，然后使用 HTML 的 link 对象。示例如下：

```
<head>
<title>文档标题</title>
<!--将 style/style.css 样式表链接到 HTML 中-->
    <link rel=stylesheet href="style/style.css" type="text/css">
</head>
```

2) 定义内部样式块对象 (Embedding a Style Block)

可以在 HTML 文档的<HTML>和<BODY>标记之间插入一个<STYLE>...</STYLE>块对象。示例如下：

```
<html>
<head>
<title>文档标题</title>
<!--CSS 样式表定义-->
<style type="text/css">
body { font: 10pt "Arial" }
h1 { font: 15pt/17pt "Arial"; font-weight: bold; color: maroon }
h2 { font: 13pt/15pt "Arial"; font-weight: bold; color: blue }
p { font: 10pt/12pt "Arial"; color: black }
</style>
</head>
...
```

注意，这里将 style 对象的 type 属性设置为“text/css”，是允许不支持这一类型的浏览器忽略样式表单。

3) 内联定义 (Inline Styles)

内联定义是在对象的标记内使用对象的 style 属性定义适用其的样式表属性。示例如下：

```
<div style="color:red;">CSS 样式表</div>    <!--将 div 中的字符设置为红色-->
```

对于 CSS 的简单介绍就到这里。下面，我们将为 9.8.1 节中的 SceneCtrl.html 添加 CSS 文件，使页面显示更具人性化。

添加的 CSS 文件为 scenectrl.css，其代码如下：

```
body {                                     /*body 样式*/
```



```

        background-color: RGB(212, 230, 234);    /*设置页面背景颜色*/
    }
    h1 {                                           /*h1 标题样式*/
        font-size: 16px;                          /*字体大小为 16 像素*/
        color:#FF3300;                           /*设置字体颜色*/
        margin-top:24px;                         /*与上方元素间隔 24 像素*/
        text-align:center;                       /*水平居中*/
    }
    h2 {                                           /*h2 标题样式*/
        font-family:"黑体";                      /*设置字体为黑体*/
        font-size:medium;                       /*字体大小中等*/
        font-weight:normal;                     /*不加粗*/
        color:#FF3300;                          /*设置字体颜色*/
        text-align:center;                      /*水平居中*/
    }
    table {                                       /*表（即 cmdtable）样式*/
        border:2px solid silver;                 /*边框宽 2 像素，为实线，银色*/
        margin:auto;                           /*table 水平居中*/
        margin-top:20px;                       /*距上方元素 20 像素*/
        width:350px;                           /*表宽 350 像素*/
        border-collapse:separate;              /*设置该表单元格之间有间隔*/
        text-align:center;                     /*table 中的文本水平居中*/
    }
    table td {                                  /*表（即 cmdtable）中的单元格样式*/
        border:1px solid silver;               /*单元格边框宽 1 像素，为实线，银色*/
        height:30px;                          /*单元格高度为 30 像素*/
    }
    #btns {                                     /*id 为“btns”的元素（即包含退出按钮的 div）的样式*/
        margin:5px auto auto auto;            /*距上方元素 5 像素，水平居中*/
        width:300px;                          /*div 宽 300 像素*/
        text-align:right;                     /*div 中的元素右对齐，即退出按钮右对齐*/
    }

```

完成了 scenectl.css 文件的创建，下面要把该样式表链接到 SceneCtrl.html 中。在 SceneCtrl.html 的 head 部分添加如下行：

```

<head>
    ...
    <title>家庭网关->家电控制->情景控制</title>
    <!--链接 scenectl.css 样式表文件-->
    <link rel="stylesheet" type="text/css" href="scenectl.css">
</head>

```

保存更改后，确保 scenectl.css 文件和 SceneCtrl.html 文件处于同一文件夹中。再次打开 SceneCtrl.html，该页面的显示效果如图 9-10 所示。

如图 9-10 所示，在添加了 CSS 样式表之后，SceneCtrl.html 有了很大的改观，布局更为

合理，也更为人性化。接下来，我们将为该HTML添加互动效果。

9.8.3 JavaScript 简介

JavaScript 是一种由 Netscape 的 LiveScript 发展而来的脚本语言，主要是为了解决服务器终端语言遗留的速度问题，将原来一些简单的但需要通过服务器完成的工作，交由 JavaScript 在客户端本地完成。例如，对用户的输入进行校验、读写 Cookie、生成动态表单，等等。

JavaScript 的正式名称是“ECMAScript”。这个标准由 ECMA（欧洲计算机厂商协会）组织发展和维护。ECMA-262 是正式的 JavaScript 标准，该标准基于 JavaScript（Netscape）和 JScript（Microsoft）。从 1996 年开始，JavaScript 出现在所有的 Netscape 和 Microsoft 浏览器中。在 1998 年，该标准成为国际 ISO 标准（ISO/IEC 16262）。这个标准目前仍在不断的发展之中。

JavaScript 使网页增加了互动性。它使有规律的、重复的 HTML 代码简化，减少了下载时间。JavaScript 能及时响应用户的操作，对提交表单做即时的校验，无须浪费时间交由服务器进行 CGI 验证。JavaScript 的特点是无穷无尽的，只要你有创意。

1. JavaScript 的基本概念

1) 运算符

运算符就是完成操作的一系列符号，共有 7 类：赋值运算符、算术运算符、比较运算符、逻辑运算符、条件运算符、位操作运算符和字符串运算符。

2) 表达式

运算符和操作数的组合称为表达式，通常分为 4 类：赋值表达式、算术表达式、布尔表达式和字符串表达式。

3) 语句

JavaScript 程序是由若干语句组成的，语句是编写程序的指令。JavaScript 提供了完整的基本编程语句，它们是：赋值语句、switch 选择语句、while 循环语句、for 循环语句、do while 循环语句、break 循环中止语句和 continue 循环中断语句。

4) 函数

函数是命名的语句段，该语句段可以被当作一个整体来引用和执行。使用函数要注意以下几点：

- (1) 函数由关键字 **function** 定义；
- (2) 函数必须先定义后使用，否则将出错；
- (3) 函数名是调用函数时引用的名称，它对大小写是敏感的，调用函数时不可写错函数名；
- (4) 参数表示传递给函数使用或操作的值，它可以是常量，也可以是变量；
- (5) **return** 语句用于返回表达式的值，也可以没有。



图 9-10 情景控制页面效果

5) 对象

JavaScript 的一个重要功能就是基于对象的功能，通过基于对象的程序设计，可以用更直观、模块化和可重复使用的方式进行程序开发。

一组包含数据的属性和对属性中所含数据进行操作的方法称为对象。例如，要设定网页的背景颜色，所针对的对象是 `document`，使用的属性名是 `bgcolor`，如 `document.bgcolor="blue"`，就是表示使背景的颜色为蓝色。

6) 事件

用户与网页交互时产生的操作称为事件。绝大部分事件都由用户的动作所引发，例如：用户按鼠标的按钮，产生 `onclick` 事件；鼠标在链接上移动，产生 `onmouseover` 事件；等等。在 JavaScript 中，事件往往与事件处理程序配套使用。

7) 变量

例如：

```
var myVariable = "some value";
```

2. 网页调用 JavaScript 的方法

HTML 调用 JavaScript 通常有两种方法。

1) 直接插入 HTML 文档

这是最常用的方法，大部分含有 JavaScript 的网页都采用这种方法，例如：

```
<script type="text/javascript">
    document.writeln("这是 JavaScript! 采用直接插入的方法!");
</script>
```

在这个例子中，我们可以看到一对新的标签：`<script>...</script>`。而 `type="text/javascript"` 用来告诉浏览器这是用 JavaScript 编写的脚本，需要调动相应的解释程序进行解释。

2) 引用脚本文件

如果已经存在一个 JavaScript 源文件（以 `js` 为扩展名），则可以采用这种引用方式，以提高程序代码的利用率。其基本格式如下：

```
<script src="script/test.js" language="Javascript"></script>
```

在本例中，HTML 将直接引用 `script` 文件夹下的 `test.js` 脚本文件，这与将 `test.js` 文件中的内容直接插入 HTML 的 `<script>` 标签中的效果是完全一样的。在实际的开发应用过程中，建议读者尽量使用这种引用文件的方法，因为这种方法可以使 HTML 文件看上去更简洁，而且 JavaScript 代码的重用性也得到了提高。

对于 JavaScript 的简要介绍先告一段落。下面我们将使用 JavaScript 进一步完善 `SceneCtrl.html` 页面的功能，使网页具有从浏览器保存的 `Cookie` 中读取信息并在网页上给出相应显示的功能。

已知，在正常情况下，用户须在另一个网页（`CtrlRoomList.html`）中选择一个要进行情景控制的房间后，才能进入情景控制网页（`SceneCtrl.html`）对该房间内的灯光设备做出相应的情景控制。

那么, 作为 SceneCtrl.html, 它如何知道用户选择了哪个房间呢? 我们采取了这样的方法, 每次当用户在房间选择页面 (CtrlRoomList.html) 中单击一个房间后, CtrlRoomList.html 中的 JavaScript 脚本会把用户当前选择的房间的房间名和房间号保存到 Cookie 中, 该 Cookie 是临时的, 在用户关闭浏览器后会被自动删除。则 SceneCtrl.html 中的 JavaScript 脚本只需读取浏览器中的 Cookie 就可以得到用户选择的房间信息了。

CtrlRoomList.html 在 Cookie 中存储的房间信息格式如下:

```
AppInfo=1,客厅;           // “1” 为房间编号, 房间名为 “客厅”
```

为了实现 SceneCtrl.html 读取 Cookie 的功能, 在 scenectrl.js 脚本文件中, 实现方式如下:

```
var roomid=0;                //全局变量, 存放当前房间的 ID
//根据 Cookie 显示房间名
function getRoomInfo()
{
    /*从 Cookie 中获取房间名和房间编号*/
    var startpos = document.cookie.indexOf("AppInfo=");
    if (startpos < 0)          //未找到相关 Cookie, 跳转到房间选择页面让用户重新选择
        window.location.replace("CtrlRoomList.html");
    else                       //找到相关 Cookie
    {
        startpos += 8;
        var strlength = document.cookie.substr(startpos).indexOf(";");
        if (strlength >= 0)
            tmpstr = document.cookie.substr(startpos, strlength);
        else tmpstr = document.cookie.substr(startpos);

        roomid=parseInt(unescape(tmpstr).split(",")[0]);    //获取房间编号
        document.getElementById("roomname").innerHTML = unescape(tmpstr).split(",")[1];
        //用从 Cookie 中获取的房间名更新 HTML 页面显示
    }
}
window.onload = getRoomInfo;    //指定当页面载入完毕后运行 getRoomInfo()函数
```

然后, 在 SceneCtrl.html 的 head 部分添加如下语句:

```
<head>
...
<!--链接 scenectrl.css 样式表文件-->
<link rel="stylesheet" type="text/css" href="scenectrl.css">
<!--引用 scenectrl.js 脚本文件-->
<script type="text/javascript" src="scenectrl.js"></script>
</head>
```

此时再次运行 SceneCtrl.html, 如果 Cookie 中存在正确的房间信息, 则页面的显示效果如图 9-11 所示。

显然, JavaScript 成功从 Cookie 中读出房间信息, 并在页面中显示出用户选择的房间名。需要指出的是, 如果没有 Cookie, 则会跳转至 CtrlRoomList.html。

9.8.4 Ajax 技术

Ajax 的全称为 “Asynchronous JavaScript and XML” (异步 JavaScript 和 XML), 是指一种创建交互式网页应用的网页开发技术。

该技术在 1998 年前后得到了应用。允许客户端脚本发送 HTTP 请求 (XMLHTTP) 的第一个组件由 Outlook Web Access 小组写成。该组件原属于微软 Exchange Server, 并且迅速地成为 Internet Explorer 4.0 的一部分。但是直到 2005 年年初, Google 在它著名的交互应用程序中使用了异步通信, 如 Google 讨论组、Google 地图、Google 搜索建议、Gmail 等, 才使 Ajax 技术真正被动态网页开发者们接受, 并予以追捧。另外, 对 Mozilla/Gecko 的支持也使得该技术走向成熟, 变得更为易用。

Ajax 技术的前景非常乐观, 它可以提高系统性能, 优化用户界面。不少第三方 Ajax 库也应运而生, 从而进一步推动了 Ajax 技术的应用、普及。

1. Ajax 的优缺点

传统的 Web 应用允许用户填写表单 (form), 当提交表单时向 Web 服务器发送一个请求。服务器接收并处理传来的表单, 然后返回一个新的网页。这个做法浪费了许多带宽, 因为在前后两个页面中的大部分 HTML 代码往往是相同的。由于每次应用的交互都需要向服务器发送请求, 因此应用的响应时间就依赖于服务器的响应时间。这导致了用户界面的响应比本地应用慢得多。

与此不同, Ajax 应用可以只向服务器发送并取回必需的数据, 它使用 SOAP 或其他一些基于 XML 的 Web Service 接口, 并在客户端采用 JavaScript 处理来自服务器的响应。因为在服务器和浏览器之间交换的数据大量减少, 结果就是我们能看到响应更快的应用; 同时, 很多的处理工作可以在发出请求的客户端机器上完成, 所以 Web 服务器的处理时间也减少了。

使用 Ajax 的最大优点, 就是能在不更新整个页面的前提下维护数据。这使得 Web 应用程序更为迅捷地回应用户动作, 并避免了在网络上发送那些没有被改变过的信息。

Ajax 不需要任何浏览器插件, 但需要用户允许 JavaScript 在浏览器上执行。就像 DHTML 应用程序那样, Ajax 应用程序必须在众多不同的浏览器和平台上经过严格的测试, 因为就目前而言, 各浏览器对 Ajax 标准的实现仍不尽相同。

目前 Ajax 最主要的问题是, 它可能破坏浏览器后退按钮的正常行为。在动态更新页面的情况下, 用户无法回到前一个页面状态, 这是因为浏览器只能记下历史记录中的静态页面。一个被完整读入的页面与一个已经被动态修改过的页面之间的差别非常微妙; 用户通常希望单击后退按钮, 就能取消他们的前一次操作, 但是在 Ajax 应用程序中却无法这样做。不过, 开发者已经想出了种种办法来解决这个问题, 其中大部分方法都是当用户单击后退按



图 9-11 从 Cookie 中读取参数后的页面效果

钮访问历史记录时，通过建立或使用一个隐藏的 IFRAME 来重现页面上的变更。

此外，由于实现 XMLHTTP 方法对系统资源要求相对较高，因此一些嵌入式设备（如手机、PDA 等）的浏览器目前还不能很好地支持 Ajax 技术。

2. Ajax 的应用方法

1) 创建 XMLHttpRequest

XMLHttpRequest 类首先由 Internet Explorer 以 ActiveX 对象引入，被称为 XMLHTTP。后来 Mozilla、Netscape、Safari 和其他浏览器也提供了 XMLHttpRequest 类，不过它们创建 XMLHttpRequest 类的方法不同。

为了在不同的浏览器下都能创建 XMLHttpRequest 以便进行 Ajax 通信，我们定义了如下函数：

```
//创建 Ajax 对象
function createAjaxRequest()
{
    ajaxobj=null;
    if (window.XMLHttpRequest)
        ajaxobj = new XMLHttpRequest();           //适用于 FireFox 等浏览器
    else if (window.ActiveXObject)
        ajaxobj = new ActiveXObject("Microsoft.XMLHTTP"); //适用于 IE
}
```

2) 发送请求

可以调用 XMLHttpRequest 类的 open()和 send()方法，如下所示：

```
ajaxobj.open('GET', URL, true);
ajaxobj.send(null);
```

open()的第一个参数是 HTTP 请求方式：GET、POST 或任何服务器所支持的用户想调用的方式。按照 HTTP 规范，该参数要大写；否则，某些浏览器（如 Firefox）可能无法处理请求。第二个参数是请求页面的 URL。第三个参数设置请求是否为异步模式。如果是 true，则 JavaScript 函数将继续执行，而不必等待服务器响应，这就是所谓的“异步”。

3) 处理服务器响应

这需要告诉 HTTP 请求对象用哪一个 JavaScript 函数处理这个响应。可以将对象的 onreadystatechange 属性设置为要使用的 JavaScript 的函数名，如下所示：

```
ajaxobj.onreadystatechange = FunctionName;
```

FunctionName 是用 JavaScript 创建的响应函数的名称。

在 Ajax 响应函数中，首先要检查请求的状态。只有当一个完整的服务器响应已经收到了，函数才可以处理该响应。XMLHttpRequest 提供了 readyState 属性来对服务器响应进行判断。readyState 的取值如下：

- 0——未初始化；
- 1——正在装载；

- 2——装载完毕;
- 3——交互中;
- 4——完成。

所以, 只有当 `readyState==4` 时, 表明一个完整的服务器响应已经收到了, 函数才可以处理该响应。具体代码如下:

```
if (ajaxobj.readyState == 4) { // 收到完整的服务器响应 }  
else { // 没有收到完整的服务器响应 }
```

当 `readyState==4` 时, 表明一个完整的服务器响应已经收到了, 接着函数会检查 HTTP 服务器响应的状态值 (`status` 属性)。完整的状态取值可参见 W3C 文档。当 HTTP 服务器响应的值为 200 时, 表示状态正常。所以, 完整的 Ajax 响应函数的框架应如下:

```
if (ajaxobj.readyState != 4) return; //响应未完成, 返回  
if (ajaxobj.status == 200) //响应完成, 且响应状态正常  
{ //对响应数据进行进一步操作 }
```

4) 处理响应数据

有以下两种方式可以得到这些数据。

(1) 以文本字符串的方式返回服务器的响应。

这是最常用的处理 Ajax 返回数据的方法。返回的文本数据位于请求对象的 `responseText` 属性中。例如, 若只想获取 CGI 脚本运行成功与否的结果, 则可以在 CGI 脚本末尾输出一句 JavaScript 语句 (如 “`success=true;`”), 在 JavaScript 脚本中接收到服务器发送回的文本数据后, 可以使用 `eval()` 关键字将接收到的文本作为 JavaScript 语句来执行。这样就能对 CGI 的运行状态做出判断, 如下所示:

```
var success=false;  
eval(ajaxobj.responseText); //将 Ajax 返回的文本作为 JavaScript 语句执行  
if (success) alert("成功! ");  
else alert("失败! ");
```

(2) 以 XML 对象方式返回服务器的响应。

这种方式要求服务器脚本返回 XML 数据流, 客户端脚本可以通过请求对象的 `responseXML` 属性获取服务器响应的 XML 数据, 并做出进一步处理。由于这种响应方式在一般的 Web 开发中并不常用, 所以这里不对其进行深入讨论。

5) Internet Explorer 浏览器的特殊情况

IE 浏览器会对每一次 Ajax 请求所返回的数据进行缓存, 一旦第二次发送相同的 Ajax 请求, IE 将首先查看缓存中是否存在相同请求返回的数据, 如果有则直接返回缓存中的数据。IE 这么做可能是为了减少与服务器之间不必要的通信, 提高页面的响应速度, 但这显然不是开发者所希望的。设想, 如果用户更改了登录密码, 当客户端发送相同的 Ajax 请求获取用户新设置的密码, 以便对用户进行验证时, IE 返回的却是上一次请求得到的旧密码, 那么这位可怜的用户可能就无法登录网站了。

要解决这个问题，其实每次只要发出不同的请求就可以了。所以一种解决方法是：在每一次的 Ajax 请求后面加上一个随机数，这样每次的请求就不同了。如下所示：

```
ajaxobj.open('GET', "password.cgi?rand=" + Math.random(), true);
ajaxobj.send(null);
```

还有一种更简便的方法，在每次 send() 前，先设置 Ajax 对象的请求头，迫使 IE 不缓存 Ajax 的返回数据。如下所示：

```
ajaxobj.open('GET', "password.cgi", true);
ajaxobj.setRequestHeader("If-Modified-Since", "0");    //设置请求头
ajaxobj.send(null);
```

下面继续完善 SceneCtrl.html，利用 Ajax 技术为其添加如下功能：当用户单击某一个情景后，利用 Ajax 向服务器发送相应的 CGI 请求，从而达到控制当前房间内所有灯光设备进入相应情景状态的目的。

JavaScript 脚本发送的 CGI 请求形如：

```
cgi-bin/scenecmd.cgi?roomid=1&cmdtype=1
```

其中，roomid 为当前房间编号，cmdtype 为情景指令索引号。

为了实现该功能，须在 scenectrl.js 脚本文件中添加如下内容：

```
var ajaxobj=null;                                //Ajax 对象，全局变量
//发送指令，onclick 事件响应函数
function sendCmd(e)
{
    if (!e) e = window.event;                    //针对 IE 浏览器，获取窗口事件对象
    var node = e.srcElement ? e.srcElement : e.target; //获取触发事件的元素

    createAjaxRequest();                          //创建 Ajax 对象，该函数在前面已提及
    ajaxobj.open("GET", "cgi-bin/scenecmd.cgi?roomid=" + roomid.toString() + "&cmdtype=" +
node.id.substr(3), true);                        //从触发事件元素的 id 中提取指令索引号
    ajaxobj.onreadystatechange = responseSendCmd; //设置回调函数
    ajaxobj.setRequestHeader("If-Modified-Since", "0"); //防止 IE Ajax 缓存问题
    ajaxobj.send(null);                           //发送 CGI 请求
}

//sendCmd()函数的 Ajax 响应函数
function responseSendCmd()
{
    if (ajaxobj.readyState != 4) return;          //响应未完成，返回
    if (ajaxobj.status == 200)                    //响应完成，且响应状态正常
    {
        var success=false;
        eval(ajaxobj.responseText)              ; //执行 CGI 返回的 JavaScript 语句
    }
}
```



```

        if (success) alert("指令发送成功！");
        else alert("指令发送失败！");
    }
}

```

为了使 JavaScript 脚本能够响应用户单击情景指令的事件，还需要在 SceneCtrl.html 中添加 onclick 事件响应，方法如下：

```

...
        <!--情景指令表-->
        <table id="cmdtable">
            <tr>
                <td id="cmd1" onclick="sendCmd(event);">全开</td>
                ...
                <td id="cmd6" onclick="sendCmd(event);">自定</td>
            </tr>
        </table>
...

```

这样就完成了对 SceneCtrl.html 页面功能的扩展，并且已经实现了该页面的大部分功能。当我们进入任一房间的情景控制页面，单击任一情景后，会有如图 9-12 所示的表现。



图 9-12 Ajax 的应用

同时，通过“串口调试助手”软件，我们截取到的 CGI 脚本的串口输出指令数据如图 9-13 所示。

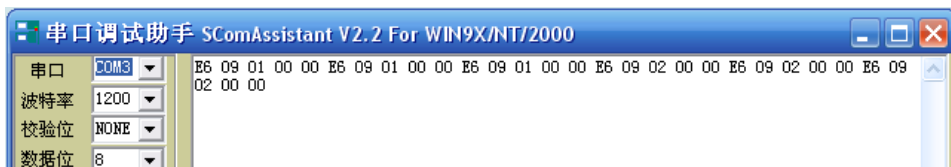


图 9-13 串口输出的数据

因为该房间内目前共设置有两个灯光设备，并且这两个灯光设备的遥控器地址不同。该房间内灯光设备的开关地址信息如图 9-14 所示。

灯光开关对码			
客厅			
开关名	遥控器地址	遥控按键号	转发器地址
test	2	1	14
测试灯光	4	7	14

图 9-14 无线遥控灯光开关对码

所以后台 CGI 脚本共发送了 6 条指令（两条记录，每条记录重复发送 3 次指令）。而且根据两个灯光开关各自的转发器地址、遥控器地址和网关指令格式不难看出，CGI 发送的指令是完全正确的。

到目前为止，我们已经掌握了网关前后台软件开发的基本流程和所需的所有基本技能，并且也实际完成了一个比较典型、完整的网关应用实例的开发。

下面我们将以网关使用说明书的形式对目前网关所具有的所有功能做一个总体的功能介绍。

9.9 网关使用说明书

9.9.1 产品概述

家庭网关是家庭网络的核心部件，是智能家居的主要部分，也是未来智能家居的主要发展方向。

家庭智能网关作为家庭网络的核心控制部件，用户可通过 Internet、手机等方式远程控制各种家用电器的运行状态，如控制电灯的开关、控制空调的温度等，并且可以设置不同的情景状态来实现对所有家电在不同情景状态下的协同控制功能。同时网关还具备安防报警功能，对于突发状况，网关会自动向用户的手机或社区的安防中心发出警报信息，并能实时抓拍现场照片，以彩信的方式发送到用户的手机上。

本网关系统的主要硬件设备包括网关控制主机、信息转发器、智能开关等。用户主要通过访问网关 Web 页面的方式来控制网关发出相应的指令，以实现家电等设备的控制与管理。

1. 产品功能

本网关具有以下功能：

- 对家中各个房间的电器设备的控制，如电视、空调、DVD、音箱、窗帘等。
- 配合智能开关，可实现对各个房间电灯的控制。

- 网关具备对码学习功能，在通过对各个家电遥控器的对码学习后即可实现网关远程控制功能。
- 具有情景设置功能。网关共预设了 6 个情景，用户单击任一情景，即可使所有进行过情景设置的灯光设备进入相应的状态。
- 用户可根据自己的实际房型进行相应的房间设置，并为每个房间添加不同的电器设备。
- 网关可以设置普通用户和超级用户两级密码，方便主人对网关核心功能的控制。

2. 使用说明

用户使用任何一台连入网络的计算机，打开浏览器，输入在家庭网关中设置的 IP 地址，即可访问网关的 Web 页面。

9.9.2 用户登录

本网关有两种用户身份——普通用户和超级用户，如图 9-15 所示，两种身份的用户权限各不相同。

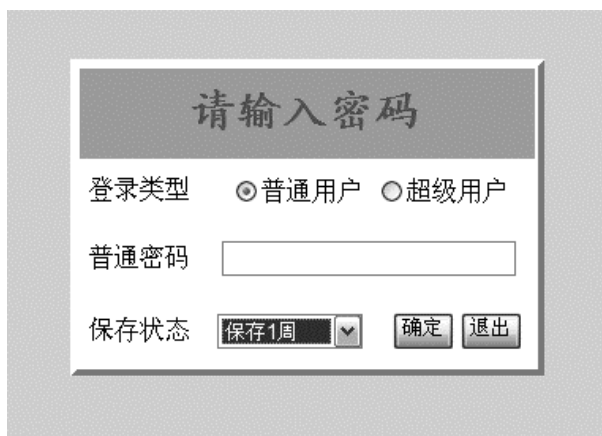
The image shows a web-based login interface for a gateway. At the top, there is a title bar with the text "请输入密码" (Please enter password). Below this, there is a section for "登录类型" (Login type) with two radio buttons: "普通用户" (General user) which is selected, and "超级用户" (Super user). Underneath, there is a label "普通密码" (General password) followed by a text input field. At the bottom, there is a "保存状态" (Save status) section with a dropdown menu currently showing "保存1周" (Save 1 week), and two buttons labeled "确定" (OK) and "退出" (Exit).

图 9-15 用户登录界面

普通用户可以访问、操作网关的普通功能，如家电控制、防盗报警及系统设置中的网络设置、对码学习等功能。

超级用户除了可以访问、操作普通用户的所有功能外，还可以进入“系统设置”→“工程维护”模块，对房间配置、超级密码等进行修改和配置。

在登录界面中，用户可以选择相应的“登录类型”，并输入相应的密码后进行登录。对于普通用户，可以选择保存登录状态的时间，这样下次登录网关时就不必再输入密码了。需要指出的是，出于安全考虑，对于超级用户，网关不提供保存登录状态的功能。

9.9.3 主界面

如图 9-16 所示，在主界面的右上角显示了用户当前的登录状态。用户可以随时选择注销登录状态，用户还可以选择相应的项目进入子菜单。



图 9-16 主界面

9.9.4 家电控制

“家电控制”模块是用户对各种家电实施远程控制的主要场所，如图 9-17 所示。



图 9-17 “家电控制”模块

1. 集中控制

单击“集中控制”后，用户需在如图 9-18 所示页面中选择需要进行家电控制的房间，单击进入。

如图 9-19 所示，进入“集中控制”菜单后，上方会显示用户选择控制的房间。用户可以单击要控制的设备进入相应的控制界面。

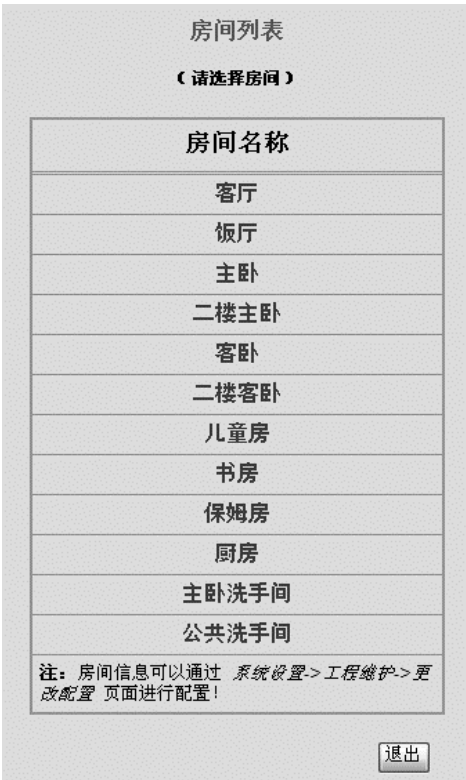


图 9-18 “房间列表”页面



图 9-19 “集中控制”菜单

1) 电视控制 (如图 9-20 所示)

在“电视控制”界面中, 如果当前房间存在电视设备, 则电视设备的名称会出现在“电视列表”中; 如果存在多台电视设备, 用户可以从列表中选择要进行控制的电视设备。

在“电视按键列表”中，凡是进行过对码学习的指令都为黑色字体，用户可以单击相应的指令，对选中的电视机进行控制。

注意：对码学习可在“系统设置”→“对码学习”中进行，下同。

空调控制、DVD 控制、功放控制和窗帘控制界面的操作方法与电视控制界面的操作方法相同，这里不再赘述。

2) 灯光控制（如图 9-21 所示）

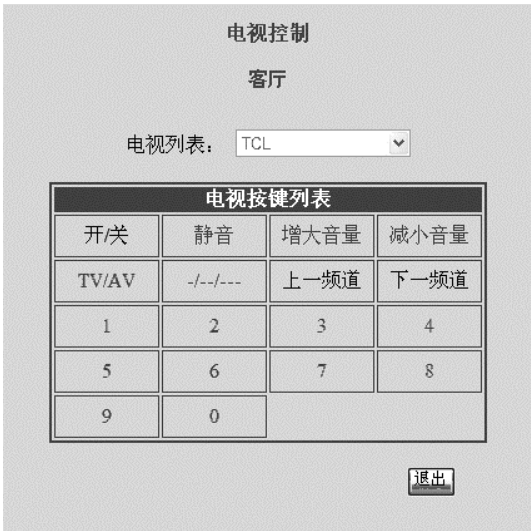


图 9-20 “电视控制”界面



图 9-21 “灯光控制”界面

在“灯光控制”界面中列出了当前房间所有已添加的灯光设备，用户可以对任一需要控制的灯光设备进行控制。

2. 情景控制

单击“情景控制”后，用户必须选择要进行情景控制的房间，才能进入“情景控制”界面。

在“情景控制”界面中有 6 种预设情景供用户选择，如图 9-22 所示。用户单击任一情景指令，便可使当前房间内所有进行过情景学习的灯光设备进入指定的情景状态。

3. 情景设置

单击“情景设置”后，用户必须选择要进行情景设置的房间，才能进入“情景设置”界面。

在“情景设置”界面中，用户可以对当前房间内的灯光设备进行情景学习，如图 9-23 所示。只有进行了情景学习后，灯光设备才会响应情景控制指令。此外，只有进行过添加或对码学习的灯光设备才能进行情景学习。

情景学习的方法是，单击“发送情景学习指令”按钮，凡是当前房间内可以被转发器控制的遥控开关全部会发出 4 声提示音，表示遥控开关进入情景学习状态。用户有 5min 的时间，将所有遥控开关置于所需的状态（开或关），然后单击该页面上的相应情景指令，所

有遥控开关会响一声，表示完成情景学习。此后，用户只要在“情景控制”界面中单击此情景状态，所有进行过情景学习的遥控开关就会进入相应的状态。如果用户在 5min 内没有发送任何情景指令，则所有的遥控开关将退出情景学习状态。



图 9-22 “情景控制”界面



图 9-23 “情景设置”界面

9.9.5 系统设置

在主菜单中单击“系统设置”即可进入系统设置模块，如图 9-24 所示。系统设置模块主要实现网关参数设置、房间配置、添加/删除设备、指令对码学习等功能。



图 9-24 系统设置模块

1. 网络设置

如图 9-25 所示，在“网络设置”界面中，用户可以设置网关的网络参数，如 IP 地址、子网掩码、默认网关、DNS 地址等。

需要指出的是，如果用户在更改了网络参数设置后出现页面长时间无响应的情况，可能是因为网关的 IP 地址等参数发生了改变所致，请用新的 IP 地址访问网关 Web 页面。

2. 密码设置

如图 9-26 所示，在“密码设置”界面中，用户可以更改普通密码和安防密码。其中，安防密码主要用于安防报警模块中，在进行布防、撤防等安防操作时使用。初始状态下，普通密码和安防密码均为“123456”。



图 9-25 “网络设置”界面

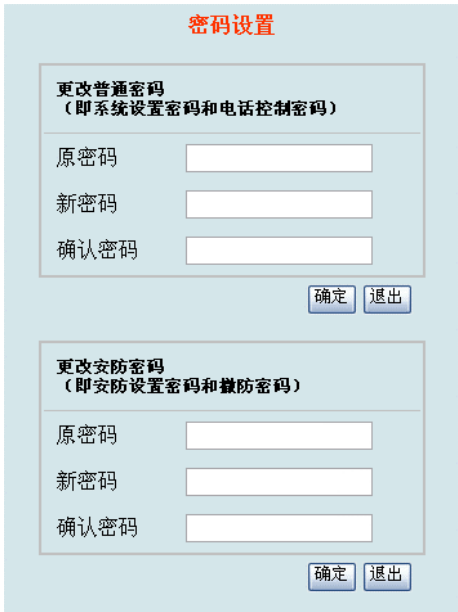


图 9-26 “密码设置”界面

3. 工程维护

只有超级用户才可以进入工程维护子菜单。如果是普通用户，系统会要求用户输入超级密码。

如图 9-27 所示，在“工程维护”模块中，用户可以进行房间配置、恢复出厂设置、配置下载等操作。



图 9-27 “工程维护”模块

1) 恢复出厂设置

如图 9-28 所示，在“恢复出厂设置”界面中，用户可以将密码、配置等恢复到出厂时的状态。

2) 修改密码

这里修改的密码是指超级密码，即进入“工程维护”模块的密码，如图 9-29 所示。



图 9-28 “恢复出厂设置”界面



图 9-29 “修改超级密码”界面

3) 更改配置

如图 9-30 所示，在“更改配置”界面中，用户可以配置房间。在这里对房间配置所做的更改将体现在“集中控制”、“情景控制”等的房间选择页面中。需要指出的是，如果删除某间房间，则该房间中所有添加的设备信息将会被全部删除。



图 9-30 “更改配置”界面

网关最多可以设置 16 个不同的房间，用户可以根据页面右边的“房间编号信息”列表将需要添加的房间编号填入房间列表中，“0”表示无此房间。

4. 对码学习

单击“对码学习”后，用户必须选择要进行对码学习的房间，才能进入“对码学习”子菜单。

如图 9-31 所示，在“对码学习”模块中，用户可以对当前房间内的电器设备进行添加、删除、对码学习等操作。



图 9-31 “对码学习”模块

1) 电视遥控对码

如图 9-32 所示，在“电视遥控对码”界面中，用户可以进行新增电视、删除现有电视、指令对码学习、清除已学习指令等操作。所有当前房间内已添加的电视设备都会显示在“电视列表”中，选择某一电视设备，就可以对该电视设备的指令进行对码学习或清除已学习指令的操作。



图 9-32 “电视遥控对码”界面

要对某一指令进行对码学习，可以单击该指令下的“学习”按钮，出现如图 9-33 所示

界面。



图 9-33 对码学习界面

根据界面提示，请在 10s 内对着网关按下电视遥控器上相应的指令按键。

如果要清除某一已学习的指令，只需单击该指令下方的“清除”按钮即可。

如果需要在当前房间内添加新的电视设备，可单击“新增电视”按钮，将出现如图 9-34 所示界面。



图 9-34 添加新的电视设备

输入新建电视的名称和转发器编号，然后单击“确定”按钮即可。

如果要删除某一电视设备，可在“电视列表”中选择要删除的电视，并单击“删除电视”按钮。需要指出的是，删除某台电视后，该电视中所有学习的指令将被全部删除。

空调遥控对码、DVD 遥控对码、功放遥控对码和窗帘遥控对码界面的操作方法与电视

遥控对码界面的操作方法相同，这里不再赘述。

2) 灯光开关对码

如图 9-35 所示，在“灯光开关对码”界面中，用户可以看到当前房间中已添加的灯光开关信息，并可以添加或删除灯光开关。



图 9-35 “灯光开关对码”界面

要添加灯光开关，可输入开关名称、遥控器地址、遥控按键号和转发器地址，然后单击“增加开关”按钮，新添加的开关信息会出现在界面上方的列表中。当然，也可以单击“进行对码”按钮，然后在 10s 内对着网关按下智能开关遥控器上的某个按键来获取遥控器地址和按键号等信息。

要删除某个灯光开关，只需单击该开关信息右边的“删除”按钮即可。

9.10 家庭网关产品的知识产权保护问题

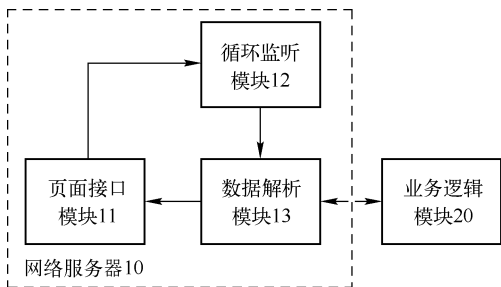
家庭网关在正式进入市场前，为了对产品进行有效的保护，防止抄袭的发生，有必要对产品申请专利，从而在知识产权的层面进行保护。近年来知识产权越来越受到重视，而知识产权分为实用新型和发明专利两种，本节将以家庭网关产品中为嵌入式网络服务技术申请专利为例，讲解实用新型专利的申请。申报专利主要涉及权利要求书和说明书，以下为具体的例子（此专利已经授权，专利号为 201020152353.0，授权公告号为 CN 201726425 U）。

说明书摘要

本实用新型涉及一种嵌入式网络服务器，包含循环连接的以下模块：与外部客户端双向连接的页面接口模块；用于侦听和接收客户端请求的循环监听模块；用于解析客户端请求的数据解析模块。数据解析模块还与家庭网关中的业务逻辑模块连接向其传递请求命令，并发送返回的请求结果数据的协议头信息至页面接口模块，由页面接口模块输出至客户端的网

络页面显示，有效地处理了来自客户端的请求信息，并返回请求结果；另外，使用单任务的服务器，通过建立请求列表来处理多路连接请求，只为 CGI 程序创建新的进程，在最大程度上节省了系统资源；同时，还通过自动生成目录、自动解压文件，使该嵌入式网络服务器具有很高的请求处理速度和效率。

摘 要 附 图



权 利 要 求 书

1. 一种嵌入式网络服务器，其特征在于，包含循环连接的以下模块：
与外部客户端双向连接的页面接口模块（11）；
用于侦听和接收客户端请求的循环监听模块（12）；
用于解析客户端请求的数据解析模块（13）。
2. 如权利要求 1 所述的嵌入式网络服务器，其特征在于，通过设置在家庭网关中，与家庭网关的业务逻辑模块（20）连接。
3. 如权利要求 2 所述的嵌入式网络服务器，其特征在于，所述数据解析模块（13）还与所述家庭网关中的业务逻辑模块（20）连接向其传递客户端请求命令，并发送返回的请求结果数据至页面接口模块（11）。
4. 如权利要求 3 所述的嵌入式网络服务器，其特征在于，所述数据解析模块（13）还根据解析出的客户端请求并根据业务逻辑模块（20）返回的数据类型的不同，通过页面接口模块（11）向客户端发送不同的 HTTP 协议头数据。
5. 如权利要求 2 所述的嵌入式网络服务器，其特征在于，所述业务逻辑模块（20）还与所述家庭网关的数据库模块（30）连接，对所述数据库模块（30）进行数据记录的检索、存取操作。
6. 如权利要求 2 所述的嵌入式网络服务器，其特征在于，所述业务逻辑模块（20）还与所述家庭网关的控制逻辑模块（40）连接，用于驱动相应的设备达到客户端要求的状态。
7. 如权利要求 1 所述的嵌入式网络服务器，其特征在于，所述页面接口模块（11）包含表单接收模块、程序返回模块，用来与客户端进行双向的数据传输；所述表单接收模块与所述循环监听模块（12）连接；所述程序返回模块与所述数据解析模块（13）连接。

8. 如权利要求 7 所述的嵌入式网络服务器, 其特征在于, 所述页面接口模块 (11) 还包含输入检查模块, 其与所述表单接收模块、所述循环监听模块 (12) 连接, 用于检查客户端提交到表单接收模块的请求数据。

说明书

嵌入式网络服务器

技术领域

本实用新型涉及一种嵌入式网络服务器, 特别涉及一种用于家庭网关的嵌入式网络服务器。

背景技术

家庭网关是智能家居系统的核心设备, 通过将家庭内部网络与外部网络 (Internet、PSTN、GPRS) 连接, 实现系统信息采集、逻辑处理、信息输出、联动控制等网络之间的信息互通。使用户能够通过计算机、PDA、手机、固定电话等通信终端实现对家庭自动化设备的远程控制, 用户还可以通过浏览器、邮件、彩信、电话等方式, 接收查询实时或历史家庭安全监控的影音图片、语音文字等多媒体信息。

家庭网关中的嵌入式网络服务器, 起到接收客户端请求、分析请求、响应请求、向客户端返回请求结果的作用。然而现有的嵌入式网络服务器在连接请求到来时, 为每个连接单独创建进程, 通过复制自身进程来多链接处理请求; 其对动态网络页面的生成是通过使用服务器脚本如 JSP、ASP 等实现的, 需要网络服务器具有这些脚本的运行支持模块。然而受嵌入式系统的资源限制, 上述数据处理过程会影响整个嵌入式网络服务器的处理速度和效率。

实用新型内容

本实用新型的目的是提供一种嵌入式网络服务器, 能够提供家庭网关与外部网络友好交互的网络页面, 能够快速有效地接收处理来自客户端的请求信息, 并返回请求结果。

为了达到上述目的, 本实用新型的技术方案是提供一种嵌入式网络服务器, 其特征在于, 包含循环连接的以下模块:

与外部客户端双向连接的页面接口模块;

用于侦听和接收客户端请求的循环监听模块;

用于解析客户端请求的数据解析模块。

上述嵌入式网络服务器, 通过设置在家庭网关中, 与家庭网关的业务逻辑模块连接。

上述数据解析模块还与上述家庭网关中的业务逻辑模块连接向其传递客户端请求命令, 并发送返回的请求结果数据至页面接口模块。

上述数据解析模块还根据解析出的客户端请求并根据业务逻辑模块返回的数据类型的不同, 通过页面接口模块向客户端发送不同的 HTTP 协议头数据。

上述业务逻辑模块还与上述家庭网关的数据库模块连接, 对上述数据库模块进行数据记录的检索、存取操作。

上述业务逻辑模块还与上述家庭网关的控制逻辑模块连接, 用于驱动相应的设备达到

客户端要求的状态。

上述页面接口模块包含表单接收模块、程序返回模块，用来与客户端进行双向的数据传输；上述表单接收模块与上述循环监听模块连接；上述程序返回模块与上述数据解析模块连接。

上述页面接口模块还包含输入检查模块，其与上述表单接收模块、上述循环监听模块连接，用于检查客户端提交到表单接收模块的请求数据。

本实用新型提供的嵌入式网络服务器，与现有技术相比，其优点在于：本实用新型设置了循环连接的页面接口模块、循环监听模块、数据解析模块，依次用于侦听和接收客户端请求并解析；数据解析模块还与家庭网关中的业务逻辑模块连接向其传递请求命令，并发送返回的请求结果数据的协议头信息至页面接口模块，由页面接口模块输出至客户端的网络页面显示，有效地处理了来自客户端的请求信息，并返回请求结果。

本实用新型在嵌入式网络服务器上使用单任务的 Boa 服务器，通过建立 HTTP 请求列表来处理多路 HTTP 连接请求，只为 CGI 程序创建新的进程，在最大程度上节省了系统资源；同时，它还通过自动生成目录、自动解压文件，使该嵌入式网络服务器具有很高的 HTTP 请求处理速度和效率。

附图说明

图 1 是本实用新型的嵌入式网络服务器与家庭网关的连接结构示意图；

图 2 是本实用新型的嵌入式网络服务器的总体结构示意图。

具体实施方式

以下结合附图说明本实用新型的具体实施方式。

请参见图 1，本实用新型提供的嵌入式网络服务器 10，设置在家庭网关中，与业务逻辑模块 20 连接。该业务逻辑模块 20 还分别与数据库模块 30 和控制逻辑模块 40 连接。

其中，网络服务器 10 用来解析客户端发出的 HTTP 请求和命令，并提交给业务逻辑模块 20；通过业务逻辑模块 20 对数据库模块 30 中的配置检测数据、日志记录等进行检索、存取等操作，并将执行指令发送给控制组件；控制组件负责驱动相应的设备至客户端要求的状态；再经由网络服务器 10 将业务逻辑模块 20 返回的数据以 XML 的形式提交给用户。

请参见图 2，嵌入式网络服务器 10 包含循环连接的页面接口模块 11、循环监听模块 12、数据解析模块 13，依次用于侦听和接收客户端请求并解析；数据解析模块 13 还与家庭网关中的业务逻辑模块 20 连接向其传递请求命令，并发送返回的请求结果数据的协议头信息至页面接口模块 11，由页面接口模块 11 输出至客户端的网络页面显示。

本实用新型中使用的嵌入式网络服务器 10 是 Boa 服务器，其是一款单任务的 HTTP 服务器，通过建立 HTTP 请求列表来处理多路 HTTP 连接请求。另外，它只为 CGI（公共网关接口）程序创建新的进程，在最大程度上节省了系统资源；同时，它还具有自动生成目录、自动解压文件等功能，使该嵌入式网络服务器 10 具有很高的 HTTP 请求处理速度和效率。

循环监听模块 12 用于网络服务器的初始化工作，包含对 Boa 进行配置，即通过在 /etc 目录下建立一个 Boa 服务器目录，在其中放入 Boa 的主要配置文件，使其能够支持 CGI 程序的执行；当有 CGI 程序请求时，循环监听模块 12 为其创建进程，并将请求输出。循环监

听模块 12 的主要工作过程如下：创建环境变量，建立侦听 TCP 流方式 SOCKET 描述符，将其转换为无阻塞套接字，并绑定 80 端口进行监听连接请求，之后进入循环侦听等待来自页面接口模块 11 的客户端请求。

数据解析模块 13 用于分析客户端请求，即将请求信息解析为方法、URL 目标、可选的查询信息及表单信息等。数据解析模块 13 与业务逻辑模块 20 配合，根据解析出的请求方法的不同，做出不同的响应：如果请求方法为 HEAD，则直接向页面接口模块 11 返回响应首部；如果请求方法为 GET，则在返回响应首部的同时，将客户端请求的 URL 目标文件从服务器目录上读出，并且发送给页面接口模块 11；如果请求方法为 POST，则将客户端发送过来的表单信息传送给相应的 CGI 程序，作为 CGI 的参数来执行 CGI 程序，并将执行结果发送给页面接口模块 11。

数据解析模块 13 还根据解析出的客户端请求并根据业务逻辑模块 20 返回的数据类型的不同（HTML 类型、纯文本、JPG 图像、GIF 图像、服务器目录下的所有文件列表信息等），通过页面接口模块 11 向客户端发送不同的 HTTP 协议头数据（文件名、大小、日期等）。

页面接口模块 11 是一段运行在网络服务器 10 上的程序，提供同客户端浏览器上 HTML 页面接口的 CGI 程序。由于在 HTML 中，表单是最主要的传递信息的手段，适用于任何浏览器，因此 CGI 程序的工作包含接收表单数据、进行数据处理，最后根据处理结果生成新的页面返回给客户端浏览器。

页面接口模块 11 包含表单接收模块、输入检查模块、程序返回模块。表单数据通常以 POST 方法提交给服务器，由表单接收模块的 CGI 程序获得，并将界面数据和内部数据对应进行下一步的处理。CGI 程序从页面获取数据是根据元素名称/值中的元素名字来进行区分的。网络服务器 10 的应用开发一般会将界面和程序逻辑脱离开来，允许页面接口模块 11 在一定程度下更改界面，如改变界面文本的属性、建立多语言版本等，而无须改动程序逻辑。表单中有很多元素，包括输入文本框、单选框、多选框、按钮等，可以提供信息的交互。

输入检查模块在用户界面上对用户提交的数据进行检查。目前一般是采用 JavaScript 脚本的方式。当提交数据时，表单对象的 onSubmit 方法会被调用，在该方法里就可以输入进行检查。常用的检查有是否必需、最大/小长度、是否字符、是否数字、E-mail 地址是否正确、IP 地址是否正确、是否匹配一个正则表达式等。

程序返回模块采用 HTML 中的注释<!--xxx-->来标记。需要在 HTML 模板中为每一个表单元素及其他任何需要程序处理的地方，按照一定规则，如注释的下一行就是表单元素行，建立其注释标记。CGI 程序可以根据注释标记来判断表单元素信息并进行处理。程序逐行读取模板文件，检查有无注释标记，如果有，则下一行需要进行处理，给表单元素赋上数据，最后就可以返回带数据的页面给客户端的浏览器。

尽管本实用新型的内容已经通过上述优选实例做了详细介绍，但应当认识到，上述的描述不应被认为是对本实用新型的限制。在本领域，技术人员阅读了上述内容后，对于本实用新型的多种修改和替代都将是显而易见的。因此，本实用新型的保护范围应由所附的权利要求来限定。

说明书附图

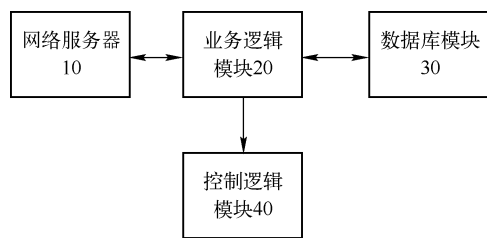


图 1

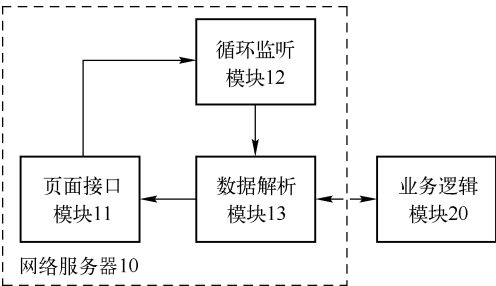


图 2

参 考 文 献

- [1] Zurawski, Richard. Embedded systems handbook. Boca Raton: CRC Press, 2009.
- [2] 王卫红, 李晓明. 计算机网络与互联网. 北京: 机械工业出版社, 2010.
- [3] 金伟正. 嵌入式 Linux 系统开发与应用. 北京: 电子工业出版社, 2011.
- [4] 陈卓, 等. 嵌入式系统开发. 北京: 电子工业出版社, 2009.
- [5] 陈赓. ARM 嵌入式技术原理与应用. 北京: 北京航空航天大学出版社, 2011.
- [6] 陈莉君. Linux 操作系统原理与应用. 北京: 清华大学出版社, 2012.
- [7] 张永忠. 嵌入式系统基础: ARM 与 Realview MDK(Keil for ARM). 北京: 北京航空航天大学出版社
- [8] Christopher A.Jones, Drew Batchelor. 张立新,等译. Linux web 编程. 北京: 电子工业出版社, 2000.
- [9] Cristian Darie, 等. 王德民, 等译. Ajax 与 PHP Web 开发. 北京: 人民邮电出版社, 2007.
- [10] Ellie Quigley. 曹晓立, 译. JavaScript 详解. 北京: 人民邮电出版社, 2011.
- [11] Patrick Griffiths. 雷钧钧, 译. HTML 之路. 北京: 机械工业出版社, 2008.
- [12] Grant Allen. 刘义宣, 译. SQLite 权威指南. 北京: 电子工业出版社, 2012.